

An Introduction to
**DATABASE
SYSTEMS**

S E V E N T H E D I T I O N

25th
ANNIVERSARY

**ACCESS THIS
BOOK ON YOUR
COMPUTER!**

*Includes a digital copy
on CD, allowing you to read
the book, take notes, and
search at any time.*

sticker: 0-201-75493-2

C . J . D A T E

Acquisitions Editor: Maite Suarez-Rivas
Associate Editor: Katherine Harutunian
Production Services: Solar Script, Inc.
Composition: Publishers' Design and Production Services, Inc.
Cover Design: Night & Day Design

Access the latest information about Addison-Wesley titles from our World Wide Web site:
<http://www.awlonline.com>

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Addison-Wesley was aware of a trademark claim, the designations have been printed in initial caps or all caps.

The programs and applications presented in this book have been included for their instructional value. They have been tested with care but are not guaranteed for any purpose. The publisher does not offer any warranties or representations, nor does it accept any liabilities with respect to the programs or applications.

This book was typeset in Quark XPress 3.3 on a Power Macintosh G3. The font families used were Times Roman and Palatino. It was printed on New Era Matte.

Library of Congress Cataloging-in-Publication Data

Date, C. J.

An introduction to database systems / by C. J. Date. — 7th ed.

p. cm.

Includes bibliographical references.

ISBN 0-201-38590-2

1. Database management. I. Title.

QA76.9.D3D3659 2000

005.74—dc21

99-30439

CIP

Copyright © 2000 by Addison Wesley Longman, Inc.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. Printed in the United States of America.

1 2 3 4 5 6 7 8 9 10-CRW-0302010099

PRELIMINARIES

Part I consists of four introductory chapters:

- Chapter 1 sets the scene by explaining what a database is and why database systems are desirable. It also briefly discusses the difference between relational database systems and others.
- Next, Chapter 2 presents a general architecture for database systems, the so-called *ANSI/SPARC architecture*. That architecture serves as a framework on which the rest of the book will build.
- Chapter 3 then presents an overview of relational systems (the aim is to serve as a gentle introduction to the much more comprehensive discussions of the same subject in Part II and later parts of the book). It also introduces and explains the running example, the suppliers and parts database.
- Finally, Chapter 4 introduces the standard relational language SQL.

An Overview of Database Management

1.1 INTRODUCTION

A **database** system is basically just a *computerized record-keeping system*. The **database** itself can be regarded as a kind of electronic filing cabinet; i.e., it is a repository or container for a collection of computerized data files. Users of the system can perform a variety of operations on such files—for example:

- Adding new, empty files to the database;
- Inserting data into existing files;
- Retrieving data from existing files;
- Changing data in existing files;
- Deleting data from existing files;
- Removing existing files from the database.

Fig. 1.1 shows a very small database containing just one file, called CELLAR, which in turn contains data concerning the contents of a wine cellar. Fig. 1.2 shows an example

BIN#	WINE	PRODUCER	YEAR	BOTTLES	READY
2	Chardonnay	Buena Vista	1997	1	1999
3	Chardonnay	Geyser Peak	1997	5	1999
6	Chardonnay	Simi	1996	4	1998
12	Joh. Riesling	Jekel	1998	1	1999
21	Fumé Blanc	Ch. St. Jean	1997	4	1999
22	Fumé Blanc	Robt. Mondavi	1996	2	1998
30	Gewurztraminer	Ch. St. Jean	1998	3	1999
43	Cab. Sauvignon	Windsor	1991	12	2000
45	Cab. Sauvignon	Geyser Peak	1994	12	2002
48	Cab. Sauvignon	Robt. Mondavi	1993	12	2004
50	Pinot Noir	Gary Farrell	1996	3	1999
51	Pinot Noir	Fetzer	1993	3	2000
52	Pinot Noir	Dehlinger	1995	2	1998
58	Merlot	Clos du Bois	1994	9	2000
64	Zinfandel	Cline	1994	9	2003
72	Zinfandel	Rafanelli	1995	2	2003

Fig. 1.1 The wine cellar database (file CELLAR)

Retrieval:		
<pre>SELECT WINE, BIN#, PRODUCER FROM CELLAR WHERE READY = 2000 ;</pre>		
Result (as shown on, e.g., a display screen):		
WINE	BIN#	PRODUCER
Cab. Sauvignon	43	Windsor
Pinot Noir	51	Fetzer
Merlot	58	Clos du Bois

Fig. 1.2 Retrieval example

of a **retrieval** operation against that database, together with the data returned by that operation. *Note:* Throughout this book we show database operations, file names, and other such material in upper case for clarity. In practice it is often more convenient to enter such material in lower case. Most systems will accept both.

Fig. 1.3 gives examples, all more or less self-explanatory, of **insert**, **change**, and **delete** operations on the wine cellar database. Examples of adding and removing entire files will be given later, in Chapters 3, 4, 5, and elsewhere.

Inserting new data:	
<pre>INSERT INTO CELLAR (BIN#, WINE, PRODUCER, YEAR, BOTTLES, READY) VALUES (53, 'Pinot Noir', 'Saintsbury', 1997, 6, 2001) ;</pre>	
Changing existing data:	
<pre>UPDATE CELLAR SET BOTTLES = 4 WHERE BIN# = 3 ;</pre>	
Deleting existing data:	
<pre>DELETE FROM CELLAR WHERE BIN# = 2 ;</pre>	

Fig. 1.3 Insert / change / delete examples

Points arising from these examples:

1. For obvious reasons, computerized files such as CELLAR in Fig. 1.1 are often called **tables** (more precisely, **relational tables**—see Sections 1.3 and 1.6).
2. The **rows** of such a table can be thought of as the records of the file, and the **columns** can be regarded as the fields of those records. In this book, we will use the terminology of records and fields when we are talking about database systems in general (mostly just in the first two chapters); we will use the terminology of rows and columns when we are talking about relational systems in particular (again, see Sections 1.3 and 1.6). *Note:* Actually, when we get to our more formal discussions in later parts of the book, we will switch to more formal terms anyway.
3. For simplicity, we have made a tacit assumption in the example that columns WINE and PRODUCER contain character string data and all other columns contain integer data. We will consider this question of column **data types** in more detail in Chapters 3, 4, and especially 5.
4. Column BIN# constitutes the **primary key** for table CELLAR (meaning no two CELLAR rows ever contain the same BIN# value). We often use *double underlining* to indicate primary key columns, as in Fig. 1.1.
5. The sample operations or “statements” in Figs. 1.2 and 1.3—**SELECT**, **INSERT**, **UPDATE**, **DELETE**—are all expressed in a language called **SQL**. SQL is the standard language for interacting with relational databases, and it is supported by just about every database product on the market today. *Note:* The name “SQL” originally stood for “Structured Query Language” and was pronounced “sequel.” Now the language has become a standard, however, the name is just a name—it is not officially an abbreviation for anything at all—and the pendulum has swung in favor of the pronunciation “ess-cue-ell.” We will assume this latter pronunciation in this book.
6. Note that SQL uses the keyword UPDATE to mean “change” specifically. This fact can cause confusion, because the term “update” is also used to refer to the three operations INSERT, UPDATE, and DELETE as a group. We will distinguish between the two meanings in this book by using lower case when the generic meaning is intended and upper case to refer to the UPDATE operator specifically.
7. As you probably already know, the vast majority of database systems in use today are in fact relational (or would-be relational, at any rate—see Chapter 4, Section 4.7). Partly for this reason, the emphasis in this book is heavily on such systems.

One last preliminary remark: An understanding of the material in this chapter and the next is fundamental to a full appreciation of the features and capabilities of a modern database system. However, it cannot be denied that the material is somewhat abstract and rather dry in places, and it does tend to involve a large number of concepts and terms that might be new to you. In later parts of the book—especially Chapters 3 and 4—you will find material that is much less abstract and hence more immediately understandable, perhaps. You might therefore prefer just to give these first two chapters a “once over lightly” reading for

now, and to reread them more carefully later as they become more directly relevant to the topics at hand.

1.2 WHAT IS A DATABASE SYSTEM?

To repeat from the previous section, a database system is basically a computerized record-keeping system; i.e., it is a computerized system whose overall purpose is to store information and to allow users to retrieve and update that information on demand. The information in question can be anything that is of significance to the individual or organization concerned—anything, in other words, that is needed to assist in the general process of running the business of that individual or organization.

Note: The terms “data” and “information” are treated as synonymous in this book. Some writers prefer to distinguish between the two, using “data” to refer to what is actually stored in the database and “information” to refer to the *meaning* of that data as understood by some user. The distinction is clearly important—so important that it seems preferable to make it explicit, where appropriate, instead of relying on a somewhat arbitrary differentiation between two essentially synonymous terms.

Fig. 1.4 is a simplified picture of a database system. It is meant to show that a database system involves four major components: **data**, **hardware**, **software**, and **users**. We con-

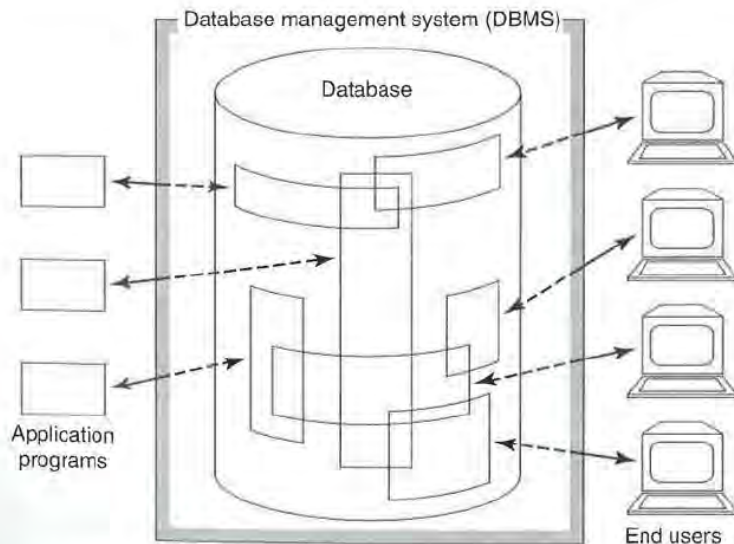


Fig. 1.4 Simplified picture of a database system

sider these four components briefly below. Later, of course, we will discuss each in much more detail (except for the hardware component, details of which are mostly beyond the scope of this book).

Data

Database systems are available on machines that range all the way from the smallest personal computers to the largest mainframes. Needless to say, the facilities provided by any given system are determined to some extent by the size and power of the underlying machine. In particular, systems on large machines ("large systems") tend to be *multi-user*, whereas those on smaller machines ("small systems") tend to be *single-user*. A **single-user system** is a system in which at most one user can access the database at any given time; a **multi-user system** is a system in which many users can access the database at the same time. As Fig. 1.4 suggests, we will normally assume the latter case in this book, for generality, but in fact the distinction is largely irrelevant so far as most users are concerned: A major objective of multi-user systems in general is precisely to allow each user to behave as if he or she were working with a *single-user* system instead. The special problems of multi-user systems are primarily problems that are internal to the system, not ones that are visible to the user (see Part IV of this book, especially Chapter 15).

Note: It is convenient to assume for the sake of simplicity that the totality of data in the system is all stored in a single database, and we will usually make that assumption in this book (since it does not materially affect any of our other discussions). In practice, however, there might be good reasons, even in a small system, why the data should be split across several distinct databases. We will touch on some of those reasons later, in Chapter 2 and elsewhere.

In general, then, the data in the database—at least in a large system—will be both *integrated* and *shared*. As we will see in Section 1.4, these two aspects, data integration and data sharing, represent a major advantage of database systems in the "large" environment; and data integration, at least, can be significant in the "small" environment as well. Of course, there are many additional advantages (to be discussed later), even in the small environment. But first let us explain what we mean by the terms *integrated* and *shared*.

- By **integrated**, we mean that the database can be thought of as a unification of several otherwise distinct files, with any redundancy among those files at least partly eliminated. For example, a given database might contain both an EMPLOYEE file, giving employee names, addresses, departments, salaries, etc., and an ENROLLMENT file, representing the enrollment of employees in training courses (refer to Fig. 1.5). Now suppose that, in order to carry out the process of training course administration, it is necessary to know the department for each enrolled student. Then there is clearly no need to include that information redundantly in the ENROLLMENT file, because it can always be discovered by referring to the EMPLOYEE file instead.
- By **shared**, we mean that individual pieces of data in the database can be shared among different users, in the sense that each of those users can have access to the same piece

EMPLOYEE	NAME	ADDRESS	DEPARTMENT	SALARY	...
ENROLLMENT	NAME	COURSE	...		

Fig. 1.5 The EMPLOYEE and ENROLLMENT files

of data, possibly for different purposes. As indicated earlier, different users can effectively even access the same piece of data *at the same time* ("concurrent access"). Such sharing, concurrent or otherwise, is partly a consequence of the fact that the database is integrated. In the example cited above, for instance, the department information in the EMPLOYEE file would typically be shared by users in the Personnel Department and users in the Education Department—and, as already suggested, those two classes of users would typically use that information for different purposes. *Note:* If the database is *not* shared, it is sometimes described as "personal" or "application-specific."

Another consequence of the foregoing facts—that the database is integrated and (usually) shared—is that any given user will typically be concerned only with some small portion of the total database; moreover, different users' portions will overlap in various ways. In other words, a given database will be perceived by different users in many different ways. In fact, even when two users share the same portion of the database, their views of that portion might differ considerably at a detailed level. This latter point is discussed more fully in Section 1.5 and Chapters 2, 3, and especially 9.

We will have more to say regarding the nature of the data component of the system in Section 1.3.

Hardware

The hardware components of the system consist of:

- The secondary storage volumes—mostly magnetic disks—that are used to hold the stored data, together with the associated I/O devices (disk drives, etc.), device controllers, I/O channels, and so forth; and
- The hardware processor(s) and associated main memory that are used to support the execution of the database system software (see the next subsection).

This book does not concern itself very much with the hardware aspects of the system, for the following reasons among others: First, those aspects form a major topic in their own right; second, the problems encountered in this area are not peculiar to database systems; and third, those problems have been very thoroughly investigated and described elsewhere.

Software

Between the physical database itself—i.e., the data as physically stored—and the users of the system is a layer of software, known variously as the **database manager** or **database server** or, most commonly, the **database management system** (DBMS). All requests for access to the database are handled by the DBMS; the facilities sketched in Section 1.1 for adding and removing files (or tables), retrieving data from and updating data in such files or tables, and so forth, are all facilities provided by the DBMS. One general function provided by the DBMS is thus *the shielding of database users from hardware-level details* (much as programming language systems shield application programmers from hardware-level details). In other words, the DBMS provides users with a perception of the database that is elevated somewhat above the hardware level, and it supports user operations (such as the SQL operations discussed briefly in Section 1.1) that are expressed in terms of that higher-level perception. We will discuss this function, and other functions of the DBMS, in considerably more detail throughout the body of this book.

A couple of further points:

- The DBMS is easily the most important software component in the overall system, but it is not the only one. Others include utilities, application development tools, design aids, report writers, and (most important) the *transaction manager* or *TP monitor*. See Chapters 2 and 3 and (especially) Part IV for further discussion of these components.
- The term *DBMS* is also used to refer generically to some particular product from some particular vendor—for example, IBM's "DB2 Universal Database" product for OS/390. The term *DBMS instance* is then sometimes used to refer to the particular copy of such a product that happens to be running on some particular computer installation. As you will surely appreciate, sometimes it is necessary to distinguish carefully between these two concepts.

Note: You should be warned that people in the industry often use the term *database* when they really mean *DBMS* (in either of the foregoing senses). Here is a typical example: "Vendor X's database outperformed vendor Y's database by a ratio of two to one." This usage is sloppy, and deprecated, but very, very common. (The problem is, of course, if we call the DBMS the database, then what do we call the database?) *Caveat lector.*

Users

We consider three broad (and somewhat overlapping) classes of users:

- First, there are **application programmers**, responsible for writing database application programs in some programming language such as COBOL, PL/I, C++, Java, or some higher-level "fourth generation" language (see Chapter 2). Such programs access the database by issuing the appropriate request—typically an SQL statement—to the DBMS. The programs themselves can be conventional batch applications, or they can be **online** applications, whose purpose is to allow an end user (see the next paragraph)

to access the database from an online workstation or terminal. Most modern applications are of the online variety.

- The second class of user, then, is **end users**, who interact with the system from online workstations or terminals. A given end user can access the database via one of the online applications mentioned in the previous paragraph, or he or she can use an interface provided as an integral part of the database system software. Such vendor-provided interfaces are also supported by means of online applications, of course, but those applications are **builtin**, not user-written. Most database systems include at least one such builtin application, namely a **query language processor**, by which the user can issue database requests (also known as statements or *commands*) such as SELECT and INSERT to the DBMS interactively. The language SQL mentioned in Section 1.1 is a typical example of a database query language.

Note: The term “query language,” common though it is, is really a misnomer, inasmuch as the natural language verb “to query” suggests *retrieval* only, whereas query languages usually (not always) provide update and other operations as well.

Most systems also provide additional builtin interfaces in which users do not issue explicit database requests such as SELECT at all, but instead operate by (e.g.) choosing items from a menu or filling in boxes on a form. Such **menu-** or **forms-driven** interfaces tend to be easier to use for people who do not have a formal training in IT (IT = information technology; the abbreviation IS, short for information systems, is also used with much the same meaning). By contrast, **command-driven interfaces**—i.e., query languages—do tend to require a certain amount of professional IT expertise, though perhaps not much (obviously not as much as is needed to write an application program in a language like COBOL). Then again, a command-driven interface is likely to be more flexible than a menu- or forms-driven one, in that query languages typically include certain features that are not supported by those other interfaces.

- The third class of user, not shown in Fig. 1.4, is the **database administrator** or DBA. Discussion of the DBA function—and the associated (very important) **data administrator** function—is deferred to Section 1.4 and Chapter 2 (Section 2.7).

This completes our preliminary description of the major aspects of a database system. We now go on to discuss the ideas in somewhat more detail.

1.3 WHAT IS A DATABASE?

Persistent Data

It is customary to refer to the data in a database as “persistent” (though it might not actually persist for very long!). By *persistent*, we mean, intuitively, that database data differs in kind from other more ephemeral data, such as input data, output data, control statements, work queues, software control blocks, intermediate results, and more generally any data that is transient in nature. More precisely, we say that data in the database “persists” because,

once it has been accepted by the DBMS for entry into the database in the first place, *it can subsequently be removed from the database only by some explicit request to the DBMS*, not as a mere side effect of (e.g.) some program completing execution. This notion of persistence thus allows us to give a slightly more precise definition of the term “database”:

- A **database** is a collection of persistent data that is used by the application systems of some given enterprise.

The term “enterprise” here is simply a convenient generic term for any reasonably self-contained commercial, scientific, technical, or other organization. An enterprise might be a single individual (with a small personal database), or a complete corporation or similar large body (with a large shared database), or anything in between. Here are some examples:

1. A manufacturing company
2. A bank
3. A hospital
4. A university
5. A government department

Any enterprise must necessarily maintain a lot of data about its operation. Such data is the “persistent data” referred to above. The enterprises just mentioned would typically include the following among their persistent data:

1. Product data
2. Account data
3. Patient data
4. Student data
5. Planning data

Note: The first few editions of this book used the term “operational data” in place of “persistent data.” That earlier term reflected the original emphasis in database systems on **operational** or **production** applications—i.e., routine, highly repetitive applications that were executed over and over again to support the day-to-day operation of the enterprise (for example, an application to support the deposit or withdrawal of cash in a banking system). The term **online transaction processing** has come to be used to refer to this kind of environment. However, databases are now increasingly being used for other kinds of applications as well—i.e., **decision support** applications—and the term “operational data” is thus no longer entirely appropriate. Indeed, enterprises nowadays often maintain two separate databases, one containing operational data and one, often called the *data warehouse*, containing decision support data. The data warehouse often includes *summary information* (e.g., totals, averages), where that summary information in turn is extracted from the operational database on a periodic basis—say once a day or once a week. See Chapter 21 for further discussion of decision support databases and applications.

Entities and Relationships

We now consider the example of a manufacturing company ("KnowWare Inc.") in a little more detail. Such an enterprise will typically wish to record information about the *projects* it has on hand; the *parts* used in those projects; the *suppliers* who supply those parts; the *warehouses* in which those parts are stored; the *employees* who work on those projects; and so on. Projects, parts, suppliers, etc., thus constitute the basic **entities** about which KnowWare Inc. needs to record information (the term "entity" is commonly used in database circles to mean any distinguishable object that is to be represented in the database). Refer to Fig. 1.6.

In addition to the basic entities themselves (suppliers, parts, and so on, in the example), there will also be **relationships** linking those basic entities together. Such relationships are represented by diamonds and connecting lines in Fig. 1.6. For example, there is a relationship ("SP") between suppliers and parts: Each supplier supplies certain parts, and conversely each part is supplied by certain suppliers (more accurately, each supplier supplies certain *kinds* of parts, each *kind* of part is supplied by certain suppliers). Similarly, parts are used in projects, and conversely projects use parts (relationship PJ); parts are stored in warehouses, and warehouses store parts (relationship WP); and so on. Note that these relationships are all *bidirectional*—that is, they can be traversed in either direction. For example, relationship SP between suppliers and parts can be used to answer both of the following questions:

- Given a supplier, get the parts supplied by that supplier;
- Given a part, get the suppliers who supply that part.

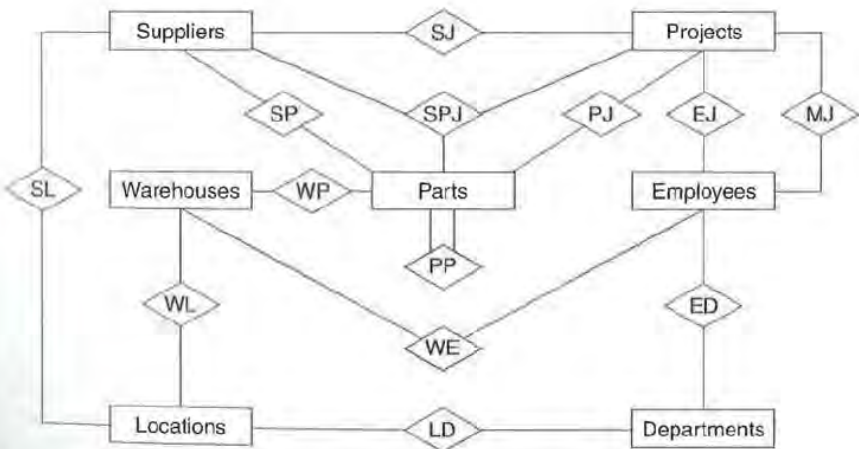


Fig. 1.6 Entity/relationship (E/R) diagram for KnowWare Inc.

The significant point about this relationship (and all of the others illustrated in the figure, of course) is that *they are just as much a part of the data as are the basic entities*. They must therefore be represented in the database, just like the basic entities. *Note:* In a relational system specifically, both the basic entities and the relationships connecting them will be represented by means of tables like the one shown in Fig. 1.1, as we will see in Chapter 3.

We note in passing that Fig. 1.6 is an example of what is called (for obvious reasons) an **entity/relationship diagram** (E/R diagram for short). In Chapter 13 we will consider such diagrams in some detail.

Fig. 1.6 also illustrates a number of other important points:

1. Although most of the relationships in that figure involve *two* entity types—i.e., they are *binary* relationships—it is by no means the case that all relationships must necessarily be binary in this sense. In the example there is one relationship (“SPJ”) involving three entity types (suppliers, parts, and projects): a *ternary* relationship. The intended interpretation is that certain suppliers supply certain parts to certain projects. Note carefully that this ternary relationship (“suppliers supply parts to projects”) is *not* equivalent, in general, to the combination of the three binary relationships “suppliers supply parts,” “parts are used in projects,” and “projects are supplied by suppliers.” For example, the statement that
 - a. Smith supplies monkey wrenches to the Manhattan project
 tells us *more* than the following three statements do:
 - b. Smith supplies monkey wrenches,
 - c. Monkey wrenches are used in the Manhattan project, and
 - d. The Manhattan project is supplied by Smith
 —we cannot (validly!) infer a. knowing only b., c., and d. More precisely, if we know b., c., and d., then we might be able to infer that Smith supplies monkey wrenches to *some* project (say project Jz), that *some* supplier (say supplier Sx) supplies monkey wrenches to the Manhattan project, and that Smith supplies *some* part (say part Py) to the Manhattan project—but we cannot validly infer that Sx is Smith or that Py is monkey wrenches or that Jz is the Manhattan project. False inferences such as these are examples of what is sometimes called **the connection trap**.
2. The figure also shows one relationship (PP) involving just *one* entity type (parts). The relationship here is that certain parts include other parts as immediate components (the so-called **bill-of-materials** relationship)—for example, a screw is a component of a hinge assembly, which is also considered as a part and might in turn be a component of some higher-level part such as a lid. Note that this relationship is still binary; it is just that the two entity types that are linked together, parts and parts, happen to be one and the same.
3. In general, a given set of entity types might be linked together in any number of distinct relationships. In the example in Fig. 1.6, there are two distinct relationships involving projects and employees: One (EJ) represents the fact that employees are assigned to projects, the other (MJ) represents the fact that employees manage projects.

We now observe that a *relationship can be regarded as an entity in its own right*. If we take as our definition of entity “any object about which we wish to record information,” then a relationship certainly fits the definition. For instance, “part P4 is stored in warehouse W8” is an entity about which we might well wish to record information—e.g., the corresponding quantity. Moreover, there are definite advantages (beyond the scope of the present chapter) to be obtained by not making any unnecessary distinctions between entities and relationships. In this book, therefore, we will usually treat relationships as just a special kind of entity.

Properties

As just indicated, an entity is any object about which we wish to record information. It follows that entities—relationships included—can be regarded as having **properties**, corresponding to the information we wish to record about them. For example, suppliers have *locations*; parts have *weights*; projects have *priorities*; assignments (of employees to projects) have *start dates*; and so on. Such properties must therefore be represented in the database. For example, the database might include a table called *S* representing suppliers, and that table might include a column called *CITY* representing supplier locations.

Properties in general can be as simple or as complex as we please. For example, the “supplier location” property is presumably quite simple, consisting as it does of just a city name, and can be represented in the database by a simple character string. By contrast, a warehouse might have a “floor plan” property, and that property might be quite complex, consisting perhaps of an entire architectural drawing and associated descriptive text. At the time of writing, most database products are only just beginning to be able to deal with complex properties such as drawings and text. We will return to this topic later in this book (especially in Chapter 5 and Part VI); until then, we will mostly assume (where it makes any difference) that properties are “simple” and can be represented by “simple” data types. Examples of such “simple” types include numbers, character strings, dates, times, and so forth.

Data and Data Models

There is another (and important) way of thinking about what data and databases really are. The word *data* derives from the Latin for “to give”; thus, data is really *given facts*, from which additional facts can be inferred. (Inferring additional facts from given facts is exactly what the DBMS is doing when it responds to a user query.) A “given fact” in turn corresponds to what logicians call a *true proposition*;^{*} for example, the statement “Supplier S1 is located in London” might be such a true proposition. It follows that a database is really *a collection of such true propositions* [1.2].

^{*}A proposition in logic is something that evaluates to either *true* or *false*, unequivocally. For instance, “William Shakespeare wrote *Pride and Prejudice*” is a proposition (a false one, as it happens).

One reason relational database systems have become so dominant—in both the industrial and the academic world, be it noted—is that they support the foregoing interpretation of data and databases very directly (almost trivially, in fact). Relational systems are based on a formal theory called **the relational model of data**, according to which:

- Data is represented by means of rows in tables, and such rows can be directly interpreted as true propositions. For example, the row for BIN# 72 in Fig. 1.1 can be interpreted in an obvious way as the following true proposition:
“Bin number 72 contains two bottles of 1995 Rafanelli Zinfandel, which will be ready to drink in 2003.”
- Operators are provided for operating on rows in tables, and those operators directly support the process of inferring additional true propositions from the given ones. As a simple example, the relational *project* operator (see Section 1.6) allows us to infer, from the true proposition just quoted, the following additional true proposition among others:
“Some bottles of Zinfandel will be ready to drink in 2003.”
(More precisely: “Some bottles of Zinfandel in some bin, produced by some producer in some year, will be ready to drink in 2003.”)

The relational model is not the only data model, however; others do exist (see Section 1.6)—though most of them differ from the relational model in that they are *ad hoc* to a degree, instead of being firmly based on formal logic. Be that as it may, the question arises: What in general is a data model? We can define the concept thus:

- A **data model** is an abstract, self-contained, logical definition of the objects, operators, and so forth, that together constitute the *abstract machine* with which users interact. The objects allow us to model the *structure* of data. The operators allow us to model its *behavior*.

We can then usefully distinguish the model from its *implementation*:

- An **implementation** of a given data model is a physical realization on a real machine of the components of the abstract machine that together constitute that model.

In a nutshell: The model is what users have to know about; the implementation is what users do not have to know about.

Note: As you can see, the distinction between model and implementation is really just a special case (a very important special case) of the familiar distinction between *logical* and *physical*. Sadly, however, many of today's database systems—even ones that profess to be relational—do not make these distinctions as clearly as they should. Indeed, there seems to be a fairly widespread lack of understanding of these distinctions and the importance of making them. As a consequence, there is all too often a gap between database *principles* (the way database systems ought to be) and database *practice* (the way they actually are). In this book we are concerned primarily with principles, but it is only fair to warn you that you might therefore be in for a few surprises, mostly of an unpleasant nature, when you start using a commercial product.

In closing this section, we should mention the fact that, actually, the term *data model* is used in the literature with two quite different meanings. The first is as described above. The second is as a model of the persistent data of some *particular enterprise* (e.g., the manufacturing company KnowWare Inc. discussed earlier in this section). The difference between the two meanings can be characterized thus:

- A data model in the first sense is like a *programming language*—albeit one that is somewhat abstract—whose constructs can be used to solve a wide variety of specific problems, but in and of themselves have no direct connection with any such specific problem.
- A data model in the second sense is like a *specific program* written in that language. In other words, a data model in the second sense takes the facilities provided by some model in the first sense and applies them to some specific problem. It can be regarded as a *specific application* of some model in the first sense.

In this book, the term *data model* will be used only in the first sense from this point forward, barring explicit statements to the contrary.

1.4 WHY DATABASE?

Why use a database system? What are the advantages? To some extent the answer to these questions depends on whether the system in question is single- or multi-user (or rather, to be more accurate, there are numerous *additional* advantages in the multi-user case). We consider the single-user case first.

Refer back to the wine cellar example once again (Fig. 1.1), which we can regard as illustrative of the single-user case. Now, that particular database is so small and so simple that the advantages might not be all that obvious. But imagine a similar database for a large restaurant, with a stock of perhaps thousands of bottles and very frequent changes to that stock; or think of a liquor store, with again a very large stock and high turnover on that stock. The advantages of a database system over traditional, paper-based methods of record-keeping are perhaps easier to see in these cases. Here are some of them:

- *Compactness*: There is no need for possibly voluminous paper files.
- *Speed*: The machine can retrieve and update data far faster than a human can. In particular, *ad hoc*, spur-of-the-moment queries (e.g., "Do we have more Zinfandel than Pinot Noir?") can be answered quickly without any need for time-consuming manual or visual searches.
- *Less drudgery*: Much of the sheer tedium of maintaining files by hand is eliminated. Mechanical tasks are always better done by machines.
- *Currency*: Accurate, up-to-date information is available on demand at any time.

The foregoing benefits apply with even more force in a multi-user environment, of course, where the database is likely to be much larger and more complex than in the

single-user case. However, there is one overriding additional advantage in such an environment, namely as follows: *The database system provides the enterprise with centralized control of its data* (which, as you should realize by now, is one of its most valuable assets). Such a situation contrasts sharply with that found in an enterprise without a database system, where typically each application has its own private files—quite often its own private tapes and disks, too—so that the data is widely dispersed and difficult to control in any systematic way.

Data Administration and Database Administration

We elaborate briefly on this concept of centralized control. The concept implies that there will be some identifiable person in the enterprise who has this central responsibility for the data. That person is the **data administrator** (DA for short) mentioned briefly at the end of Section 1.2. Given that (to repeat) the data is one of the enterprise's most valuable assets, it is imperative that there should be some person who understands that data, and the needs of the enterprise with respect to that data, *at a senior management level*. The data administrator is that person. Thus, it is the data administrator's job to decide what data should be stored in the database in the first place, and to establish policies for maintaining and dealing with that data once it has been stored. An example of such a policy might be one that dictates who can perform what operations on what data in what circumstances—in other words, a *data security* policy (see the next subsection).

Note carefully that the data administrator is a manager, not a technician (although he or she certainly does need to have some appreciation of the capabilities of database systems at a technical level). The *technical* person responsible for implementing the data administrator's decisions is the **database administrator** (DBA for short). The DBA, unlike the data administrator, is thus an *IT professional*. The job of the DBA is to create the actual database and to implement the technical controls needed to enforce the various policy decisions made by the data administrator. The DBA is also responsible for ensuring that the system operates with adequate performance and for providing a variety of other technical services. The DBA will typically have a staff of system programmers and other technical assistants (i.e., the DBA function will typically be performed in practice by a team of people, not just by one person); for simplicity, however, it is convenient to assume that the DBA is indeed a single individual. We will discuss the DBA function in more detail in Chapter 2.

Benefits of the Database Approach

In this subsection we identify some of the specific advantages that accrue from the foregoing notion of centralized control.

- *The data can be shared*

We discussed this point in Section 1.2, but for completeness we mention it again here. Sharing means not only that existing applications can share the data in the database, but also that new applications can be developed to operate against that same data. In other

words, it might be possible to satisfy the data requirements of new applications without having to add any new data to the database.

■ *Redundancy can be reduced*

In nondatabase systems each application has its own private files. This fact can often lead to considerable redundancy in stored data, with resultant waste in storage space. For example, a personnel application and an education records application might both own a file that includes department information for employees. As suggested in Section 1.2, however, those two files can be integrated, and the redundancy eliminated, so long as the data administrator is aware of the data requirements for both applications—i.e., so long as the enterprise has the necessary overall control.

Incidentally, we do not mean to suggest that *all* redundancy can or necessarily should be eliminated. Sometimes there are sound business or technical reasons for maintaining several distinct copies of the same data. However, we do mean to suggest that any such redundancy should be carefully **controlled**; that is, the DBMS should be aware of it, if it exists, and should assume responsibility for “propagating updates” (see the next point below).

■ *Inconsistency can be avoided (to some extent)*

This is really a corollary of the previous point. Suppose a given fact about the real world—say the fact that employee E3 works in department D8—is represented by two distinct entries in the database. Suppose also that the DBMS is not aware of this duplication (i.e., the redundancy is not controlled). Then there will necessarily be occasions on which the two entries will not agree: namely, when one of the two has been updated and the other not. At such times the database is said to be *inconsistent*. Clearly, a database that is in an inconsistent state is capable of supplying incorrect or contradictory information to its users.

Of course, if the given fact is represented by a single entry (i.e., if the redundancy is removed), then such an inconsistency cannot occur. Alternatively, if the redundancy is not removed but is *controlled* (by making it known to the DBMS), then the DBMS can guarantee that the database is never inconsistent *as seen by the user*, by ensuring that any change made to either of the two entries is automatically applied to the other one as well. This process is known as **propagating updates**.

■ *Transaction support can be provided*

A **transaction** is a logical unit of work, typically involving several database operations (in particular, several update operations). The standard example involves the transfer of a cash amount from one account *A* to another account *B*. Clearly two updates are required here, one to withdraw the cash from account *A* and the other to deposit it to account *B*. If the user has stated that the two updates are part of the same transaction, then the system can effectively guarantee that either both of them are done or neither is—even if, e.g., the system fails (say because of a power outage) halfway through the process.

Note: The transaction *atomicity* feature just illustrated is not the only benefit of transaction support, but unlike some of the others it is one that applies even in the

single-user case.* A full description of all of the various advantages of transaction support and how they can be achieved appears in Chapters 14 and 15.

■ *Integrity can be maintained*

The problem of integrity is the problem of ensuring that the data in the database is correct. Inconsistency between two entries that purport to represent the same "fact" is an example of lack of integrity (see the discussion of this point earlier in this subsection); of course, this particular problem can arise only if redundancy exists in the stored data. Even if there is no redundancy, however, the database might still contain incorrect information. For example, an employee might be shown as having worked 400 hours in the week instead of 40, or as belonging to a department that does not exist. Centralized control of the database can help in avoiding such problems—insofar as they can be avoided—by permitting the data administrator to define, and the DBA to implement, **integrity constraints** (also known as *business rules*) to be checked whenever any update operation is performed.

It is worth pointing out that data integrity is even more important in a database system than it is in a "private files" environment, precisely because the data is shared. For without appropriate controls it would be possible for one user to update the database incorrectly, thereby generating bad data and so "infecting" other innocent users of that data. It should also be mentioned that most database products are currently rather weak in their support for integrity constraints (though there have been some recent improvements in this area). This fact is unfortunate, given that—as we will see in Chapter 8—integrity constraints are both fundamental and crucially important, much more so than is usually realized.

■ *Security can be enforced*

Having complete jurisdiction over the database, the DBA (under appropriate direction from the data administrator, of course) can ensure that the only means of access to the database is through the proper channels, and hence can define **security constraints** or rules to be checked whenever access is attempted to sensitive data. Different constraints can be established for each type of access (retrieve, insert, delete, etc.) to each piece of information in the database. Note, however, that without such constraints the security of the data might actually be more at risk than in a traditional (dispersed) filing system; that is, the centralized nature of a database system in a sense *requires* that a good security system be in place also.

■ *Conflicting requirements can be balanced*

Knowing the overall requirements of the enterprise (as opposed to the requirements of individual users), the DBA—again under the data administrator's direction—can so structure the system as to provide an overall service that is "best for the enterprise." For

*On the other hand, single-user systems often do not provide transaction support at all but simply leave the problem on the user's shoulders.

example, a physical representation can be chosen for the data in storage that gives fast access for the most important applications (possibly at the cost of slower access for certain other applications).

- *Standards can be enforced*

With central control of the database, the DBA (under the direction of the data administrator once again) can ensure that all applicable standards are observed in the representation of the data. Applicable standards might include any or all of the following: departmental, installation, corporate, industry, national, and international standards. Standardizing data representation is particularly desirable as an aid to *data interchange*, or movement of data between systems (this consideration is becoming particularly important with the advent of distributed systems—see Chapters 2 and 20). Likewise, data naming and documentation standards are also very desirable as an aid to data sharing and understandability.

Most of the advantages listed above are probably fairly obvious. However, one further point, which might not be so obvious (although it is in fact implied by several of the others) needs to be added to the list: namely, *the provision of data independence*. (Strictly speaking, this is an *objective* for database systems, rather than an advantage necessarily.) The concept of data independence is so important that we devote a separate section to it.

1.5 DATA INDEPENDENCE

We begin by observing that there are two kinds of data independence, physical and logical [1.3–1.4]; for the time being, however, we will be concerned with the physical kind only. Until further notice, therefore, the unqualified term “data independence” should be understood to mean *physical* data independence specifically; we will discuss logical data independence in Chapters 2, 3, and especially 9. *Note:* We should perhaps say too that the term “data independence” is not very apt (it does not capture very well the essence of what is really going on); however, it is the term traditionally used, and we will stay with it in this book.

Data independence can most easily be understood by first considering its opposite. Applications implemented on older systems—prerelational or even predatabase systems—tend to be *data-dependent*. What this means is that the way the data is physically represented in secondary storage, and the technique used to access it, are both dictated by the requirements of the application under consideration, and moreover that *knowledge of that physical representation and that access technique is built into the application code*.

- *Example:* Suppose we have an application that uses the EMPLOYEE file of Fig. 1.5, and suppose it is decided, for performance reasons, that the file is to be indexed on its “employee name” field. In an older system, the application in question will typically be aware of the fact that the index exists, and aware also of the record sequence as defined by that index, and the internal structure of the application will be built around that knowledge. In particular, the precise form of the various data access and exception

checking routines within the application will depend very heavily on details of the interface presented to the application by the data management software.

We say that an application such as the one in this example is **data-dependent**, because it is impossible to change the physical representation (how the data is physically represented in storage) or access technique (how it is physically accessed) without affecting the application, probably drastically. For instance, it would not be possible to replace the index in the example by a hashing scheme without making major modifications to the application. What is more, the portions of the application requiring alteration in such a situation are precisely those portions that communicate with the data management software; the difficulties involved are quite irrelevant to the problem the application was originally written to solve—i.e., they are difficulties *introduced* by the nature of the data management interface.

In a database system, however, it would be extremely undesirable to allow applications to be data-dependent in the foregoing sense, for at least the following two reasons:

1. Different applications will require different views of the same data. For example, suppose that before the enterprise introduces its integrated database, there are two applications, A and B, each owning a private file that includes the field “customer balance.” Suppose, however, that application A stores that field in decimal, whereas application B stores it in binary. It will still be possible to integrate the two files, and to eliminate the redundancy, provided the DBMS is ready and able to perform all necessary conversions between the stored representation chosen (which might be decimal or binary or something else again) and the form in which each application wishes to see it. For example, if it is decided to store the field in decimal, then every access by B will require a conversion to or from binary.

This is a fairly trivial example of the kind of difference that might exist in a database system between the data as seen by a given application and the data as physically stored. We will consider many other possible differences later in this section.

2. The DBA must have the freedom to change the physical representation or access technique in response to changing requirements, without having to modify existing applications. For example, new kinds of data might be added to the database; new standards might be adopted; application priorities (and therefore relative performance requirements) might change; new storage devices might become available; and so on. If applications are data-dependent, such changes will typically require corresponding changes to programs, thus tying up programmer effort that would otherwise be available for the creation of new applications. It is still not uncommon, even today, to find a significant fraction of the available programming effort devoted to this kind of maintenance (think of the “Year 2000” problem!)—hardly the best use of a scarce and valuable resource.

It follows that the provision of data independence is a major objective for database systems. Data independence can be defined as **the immunity of applications to change in physical representation and access technique**—which implies, of course, that the applications concerned do not depend on any one particular physical representation or access technique. In Chapter 2, we describe an architecture for database systems that provides a

basis for achieving this objective. Before then, however, let us consider in more detail some examples of the types of changes that the DBA might wish to make, and that we might therefore wish applications to be immune to.

We start by defining three terms: *stored field*, *stored record*, and *stored file* (refer to Fig. 1.7).

- A **stored field** is, loosely, the smallest unit of stored data. The database will contain many **occurrences** (or **instances**) of each of several **types** of stored field. For example, a database containing information about different kinds of parts might include a stored field type called "part number," and then there would be one occurrence of that stored field for each kind of part (screw, hinge, lid, etc.).

Note: In practice it is common to drop the qualifiers "type" and "occurrence" and to rely on context to indicate which is meant. Although there is a slight risk of confu-

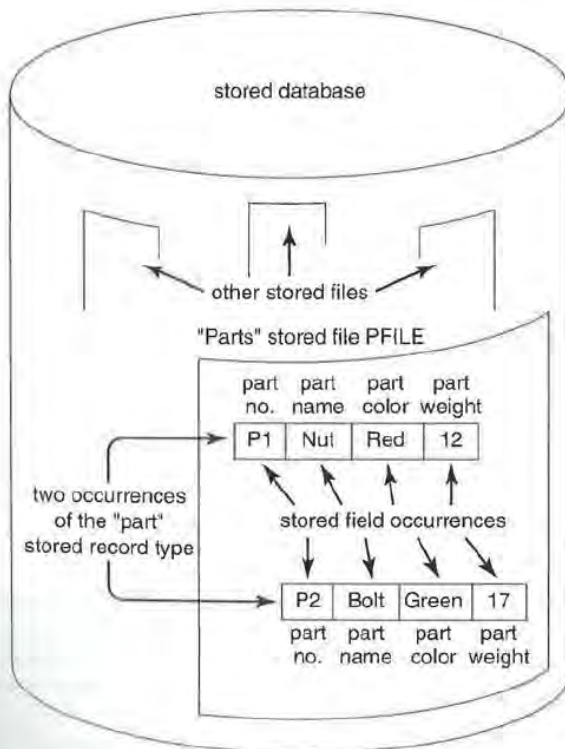


Fig. 1.7: Stored fields, records, and files

sion, the practice is convenient, and we will adopt it ourselves from time to time in this book. (These remarks apply to stored records as well—see the paragraph immediately following.)

- A **stored record** is a collection of related stored fields. Again we distinguish between type and occurrence. A stored record **occurrence** (or **instance**) consists of a group of related stored field occurrences. For example, a stored record occurrence in the “parts” database might consist of an occurrence of each of the following stored fields: part number, part name, part color, and part weight. We say that the database contains many occurrences of the “part” stored record **type** (again, one occurrence for each kind of part).
- Finally, a **stored file** is the collection of all currently existing occurrences of one type of stored record. *Note:* We assume for simplicity that any given stored file contains just one type of stored record. This simplification does not materially affect any of our subsequent discussions.

Now, in nondatabase systems it is normally the case that any given *logical* record as seen by some application is identical to some corresponding *stored* record. As we have already seen, however, this is not necessarily the case in a database system, because the DBA might need to be able to make changes to the stored representation of data—that is, to the stored fields, records, and files—while the data as seen by applications does *not* change. For example, the DEPARTMENT field in the EMPLOYEE file might be stored in binary to economize on storage space, whereas a given COBOL application might see it as a character string. And later the DBA might decide for some reason to change the stored representation of that field from binary to decimal, say, and yet still allow the COBOL application to see it in character form.

As stated earlier, a difference such as this one, involving data type conversion on some field on each access, is comparatively minor. In general, however, the difference between what the application sees and what is actually stored might be quite considerable. To amplify this remark, we present below a list of aspects of the stored representation that might be subject to change. You should consider in each case what the DBMS would have to do to make applications immune to such change (and indeed whether such immunity can always be achieved).

- *Representation of numeric data*

A numeric field might be stored in internal arithmetic form (e.g., packed decimal) or as a character string. Either way, the DBA must choose an appropriate base (e.g., binary or decimal), scale (fixed or floating point), mode (real or complex), and precision (number of digits). Any of these aspects might need to be changed to improve performance or to conform to a new standard or for many other reasons.

- *Representation of character data*

A character string field might be stored using any of several distinct coded character sets—e.g., ASCII, EBCDIC, Unicode.

■ Units for numeric data

The units in a numeric field might change—from inches to centimeters, for example, during a process of metrication.

■ Data coding

In some situations it might be desirable to represent data in storage by coded values. For example, the “part color” field, which an application sees as a character string (“Red” or “Blue” or “Green” ...), might be stored as a single decimal digit, interpreted according to the coding scheme 1 = “Red,” 2 = “Blue,” and so on.

■ Data materialization

In practice the *logical* field as seen by an application does usually correspond to some specific stored field (although, as we have already seen, there might be differences in data type, coding, and so on). In such a case, the process of materialization—that is, constructing an occurrence of the logical field from the corresponding stored field occurrence and presenting it to the application—can be said to be *direct*. Sometimes, however, a logical field will have no single stored counterpart; instead, its values will be materialized by means of some computation, perhaps on several stored field occurrences. For example, values of the logical field “total quantity” might be materialized by summing a number of individual stored quantities. “Total quantity” here is an example of a **virtual** field, and the materialization process is said to be *indirect*. Note, however, that the user might see a difference between real and virtual fields, inasmuch as it might not be possible to update an occurrence of a virtual field (at least, not directly).

■ Structure of stored records

Two existing stored records might be combined into one. For example, the stored records

part no.	part color	and	part no.	part weight
----------	------------	-----	----------	-------------

might be combined to form

part no.	part color	part weight
----------	------------	-------------

Such a change might occur as existing applications are integrated into the database system. It implies that an application’s logical record might consist of a proper subset of the corresponding stored record—that is, certain fields in that stored record would be invisible to the application in question.

Alternatively, a single stored record type might be split into two. Reversing the previous example, the stored record

part no.	part color	part weight
----------	------------	-------------

might be split into

part no.	part color	and	part no.	part weight
----------	------------	-----	----------	-------------

Such a split would allow less frequently used portions of the original record to be stored on a slower device, for example. The implication is that an application's logical record might contain fields from several distinct stored records—that is, it might be a proper superset of any given one of those stored records.

■ *Structure of stored files*

A given stored file can be physically implemented in storage in a wide variety of ways. For example, it might be entirely contained within a single storage volume (e.g., a single disk), or it might be spread across several volumes (possibly on several different device types); it might or might not be physically sequenced according to the values of some stored field; it might or might not be sequenced in one or more additional ways by some other means, e.g., by one or more indexes or one or more embedded pointer chains (or both); it might or might not be accessible via some hashing scheme; the stored records might or might not be physically blocked; and so on. But none of these considerations should affect applications in any way (other than performance, of course).

This concludes our list of aspects of the stored data representation that are subject to possible change. The list implies (among other things) that the database should be able to **grow** without impairing existing applications; indeed, enabling the database to grow without logically impairing existing applications is one of the most important reasons for requiring data independence in the first place. For example, it should be possible to extend an existing stored record by the addition of new stored fields, representing, typically, further information concerning some existing type of entity (e.g., a “unit cost” field might be added to the “part” stored record). Such new fields should simply be invisible to existing applications. Likewise, it should be possible to add entirely new stored record types (and hence new stored files), again without requiring any change to existing applications; such records would typically represent new entity types (e.g., a “supplier” record type could be added to the “parts” database). Again, such additions should be invisible to existing applications.

By now you might have realized that data independence is one of the reasons why separating the data model from its implementation, as discussed near the end of Section 1.3, is so important: To the extent we do not make that separation, we will not achieve data independence. The widespread failure to make the separation properly, in today's SQL systems in particular, is thus particularly distressing. *Note:* We do not mean to suggest by these remarks that today's SQL systems provide no data independence at all, only that they do provide much less than relational systems are theoretically capable of. In other words, data independence is not an absolute (different systems provide it to different degrees, and few if any provide none at all); SQL systems provide more than older systems, but they are still not perfect, as we will see in the chapters to come.

1.6 RELATIONAL SYSTEMS AND OTHERS

As mentioned at the end of Section 1.3, DBMS products that are based on the *relational model of data* ("relational systems") have come to dominate the database marketplace. What is more, the vast majority of database research over the last 30 years has also been based—albeit a little indirectly, in some cases—on that model. In fact, the introduction of the relational model in 1969–70 was undeniably the single most important event in the entire history of the database field. For these reasons, plus the additional reason that the relational model is solidly based on logic and mathematics and therefore provides an ideal vehicle for teaching database principles, the emphasis in this book is (as noted in Section 1.1) heavily on relational systems.

What then exactly is a relational system? It is obviously not possible to answer this question fully at this early point in the book—but it is possible, and desirable, to give a rough and ready answer, which we can make more precise later. Briefly (and loosely), a relational system is one in which:

1. The data is perceived by the user as tables (and nothing but tables); and
2. The operators available to the user (e.g., for retrieval) are operators that generate new tables from old. For example, there is one operator, *restrict*, which extracts a subset of the rows of a given table, and another, *project*, which extracts a subset of the columns—and of course a row subset and a column subset of a table can both in turn be regarded as tables themselves, as we will see in just a moment.

Note: The reason such systems are called "relational" is that the term *relation* is basically just a mathematical term for a *table*; indeed, the terms *relation* and *table* can be taken as synonymous, at least for informal purposes (see Chapters 3 and 5 for further discussion). Perhaps we should add that the reason is definitely *not* that the term *relation* is "basically just a mathematical term for" a *relationship* in the sense of entity/relationship diagrams (see Section 1.3); in fact, there is very little direct connection between relational systems and such diagrams, as we will see in Chapter 13.

To repeat, we will make the foregoing definitions much more precise later, but they will serve for the time being. Fig. 1.8 provides an illustration. The data—see part a. of the figure—consists of a single table, named *CELLAR* (in fact, it is a scaled down version of the *CELLAR* table from Fig. 1.1, reduced in size to make it more manageable). Two sample retrievals—one involving a *restriction* or row-subsetting operation and the other a *projection* or column-subsetting operation—are shown in part b. of the figure. *Note:* The two retrievals are expressed in SQL once again.

We can now distinguish between relational and nonrelational systems, as follows. As already stated, the user of a relational system sees the data as tables, and nothing but tables. The user of a nonrelational system, by contrast, sees other data structures, either instead of or in addition to the tables of a relational system. Those other structures, in turn, require other operators to manipulate them. For example, in a *hierarchical* system such as IBM's IMS, the data is represented to the user in the form of a set of tree structures (hierarchies),

a. Given table:		CELLAR	WINE	YEAR	BOTTLES
			Chardonnay	1996	4
			Fumé Blanc	1996	2
			Pinot Noir	1993	3
			Zinfandel	1994	9

b. Operators (examples):					
1. Restrict:		Result:	WINE	YEAR	BOTTLES
SELECT WINE, YEAR, BOTTLES			Chardonnay	1996	4
FROM CELLAR			Fumé Blanc	1996	2
WHERE YEAR > 1995 ;					

2. Project:		Result:	WINE	BOTTLES
SELECT WINE, BOTTLES			Chardonnay	4
FROM CELLAR ;			Fumé Blanc	2
			Pinot Noir	3
			Zinfandel	9

Fig. 1.8 Data structure and operators in a relational system (examples)

and the operators provided for manipulating such structures include operators for *traversing pointers*—viz., the pointers that represent the hierarchic paths up and down the trees. (By contrast, it is an important distinguishing characteristic of relational systems that they involve no such pointers, as we have seen.)

To pursue the issue a little further: Database systems can in fact be conveniently categorized according to the data structures and operators they present to the user. According to this scheme, the oldest (prerelational) systems fall into three broad categories, namely **inverted list**, **hierarchic**, and **network** systems.* We do not discuss these categories in detail in this book because from a technological point of view, at least, they must be regarded as obsolete. (You can find tutorial descriptions of all three in reference [1.5] if you are interested.) However, we should at least say that the term *network* in this context has nothing to do with a *communications network*!—it refers rather (to repeat) to the kinds of data structures and operators supported by the systems in question.

*By analogy with the relational model, earlier editions of this book referred to inverted list, hierarchic, and network *models* (and much of the literature still does). To talk in such terms is rather misleading, however, because, unlike the relational model, the inverted list, hierarchic, and network “models” were all defined *after the fact*; i.e., commercial inverted list, hierarchic, and network products were implemented *first*, and the corresponding “models” were defined *subsequently* by a process of induction—in this context, a polite term for guesswork—from those existing implementations.

Note: Network systems are sometimes called either **CODASYL** systems or **DBTG** systems, after the body that proposed them—namely, the Data Base Task Group (DBTG) of the Conference on Data Systems Languages (CODASYL). Probably the best known example of such a system is IDMS, from Computer Associates International Inc. Like hierarchical systems (but unlike relational ones), all such systems expose pointers to the user (among other things).

The first **relational** products began to appear in the late 1970s and early 1980s. At the time of writing, the vast majority of database systems are relational, and they run on just about every kind of hardware and software platform available. Leading examples include, in alphabetical order, DB2 (various versions) from IBM Corp.; Ingres II from Computer Associates International Inc.; Informix Dynamic Server from Informix Software Inc.; Microsoft SQL Server from Microsoft Corp.; Oracle 8i from Oracle Corp.; and Sybase Adaptive Server from Sybase Inc. *Note:* When we have cause to refer to any of these products later in this book, we will refer to them (as most of the industry does, informally) by the abbreviated names DB2, Ingres (pronounced “ingress”), Informix, SQL Server, Oracle, and Sybase, respectively.

More recently, certain **object** and **object/relational** products have become available.* The object/relational systems represent (for the most part) upward-compatible extensions to certain of the original relational products, as in the case of DB2 and Informix; the object (sometimes *object-oriented*) systems represent attempts to do something entirely different, as in the case of GemStone from GemStone Systems Inc. and Versant ODBMS from Versant Object Technology. We will discuss these newer systems in Part VI of this book.

In addition to the various approaches mentioned above, research has proceeded over the years on a variety of alternative schemes, including the **multi-dimensional** approach and the **logic-based** (also called *deductive* or *expert*) approach. We discuss multi-dimensional systems in Chapter 21 and logic-based systems in Chapter 23.

1.7 SUMMARY

We close this introductory chapter by summarizing the main points discussed. First, a **database system** can be thought of as a computerized record-keeping system. Such a system involves the **data** itself (stored in the **database**), **hardware**, **software** (in particular the **database management system** or DBMS), and—most important!—**users**. Users in turn can be divided into **application programmers**, **end users**, and the **database administrator** or DBA. The DBA is responsible for administering the database and database system in accordance with policies established by the **data administrator**.

Databases are **integrated** and (usually) **shared**; they are used to store **persistent data**. Such data can usefully, albeit informally, be considered as representing **entities**, together with **relationships** among those entities—although in fact a relationship is really just a special kind of entity. We very briefly examined the idea of **entity/relationship diagrams**.

*The term *object* here has a rather specific meaning, which we will explain in Part VI. Prior to that point, we will use the term in its normal generic sense, barring explicit statements to the contrary.

Database systems provide a number of benefits, of which one of the most important is (physical) **data independence**. Data independence can be defined as the immunity of application programs to changes in the way the data is physically stored and accessed. Among other things, data independence requires that a sharp distinction be made between the **data model** and its **implementation**. (We remind you in passing that the term *data model*, perhaps unfortunately, has two rather different meanings.)

Database systems also usually support **transactions** or logical units of work. One advantage of transactions is that they are guaranteed to be **atomic** (all or nothing), even if the system fails in the middle of their execution.

Finally, database systems can be based on a number of different approaches. Relational systems in particular are based on a formal theory called **the relational model**, according to which data is represented as rows in tables (interpreted as **true propositions**) and operators are provided that directly support the process of **inferring** additional true propositions from the given ones. From both an economic and a theoretical perspective, relational systems are easily the most important (and this state of affairs is not likely to change in the foreseeable future). We have seen a few simple examples of **SQL**, the standard language for dealing with relational systems (in particular, examples of the **SQL SELECT, INSERT, UPDATE, and DELETE** statements). This book will be heavily based on relational systems, although—for reasons explained in the preface—not so much on *SQL per se*.

EXERCISES

1.1 Define the following terms:

binary relationship	menu-driven interface
command-driven interface	multi-user system
concurrent access	online application
data administration	persistent data
database	property
database system	query language
data independence	redundancy
DBA	relationship
DBMS	security
entity	sharing
entity/relationship diagram	stored field
forms-driven interface	stored file
integration	stored record
integrity	transaction

1.2 What are the advantages of using a database system?

1.3 What are the disadvantages of using a database system?

1.4 What do you understand by the term *relational system*? Distinguish between relational and non-relational systems.

1.5 What do you understand by the term *data model*? Explain the difference between a data model and its implementation. Why is the difference important?

1.6 Show the effects of the following SQL retrieval operations on the wine cellar database of Fig. 1.1:

- a.

```
SELECT WINE, PRODUCER
FROM CELLAR
WHERE BIN# = 72 ;
```
- b.

```
SELECT WINE, PRODUCER
FROM CELLAR
WHERE YEAR > 1996 ;
```
- c.

```
SELECT BIN#, WINE, YEAR
FROM CELLAR
WHERE READY < 1999 ;
```
- d.

```
SELECT WINE, BIN#, YEAR
FROM CELLAR
WHERE PRODUCER = 'Robt. Mondavi'
AND BOTTLES > 6 ;
```

1.7 Give in your own words an interpretation as a true proposition of a typical row from each of your answers to Exercise 1.6.

1.8 Show the effects of the following SQL update operations on the wine cellar database of Fig. 1.1:

- a.

```
INSERT
INTO CELLAR ( BIN#, WINE, PRODUCER, YEAR, BOTTLES, READY )
VALUES ( 80, 'Syrah', 'Meridian', 1994, 12, 1999 ) ;
```
- b.

```
DELETE
FROM CELLAR
WHERE READY > 2000 ;
```
- c.

```
UPDATE CELLAR
SET BOTTLES = 5
WHERE BIN# = 50 ;
```
- d.

```
UPDATE CELLAR
SET BOTTLES = BOTTLES + 2
WHERE BIN# = 50 ;
```

1.9 Write SQL statements to perform the following operations on the wine cellar database:

- a. Get bin number, name of wine, and number of bottles for all Geyser Peak wines.
- b. Get bin number and name of wine for all wines for which there are more than five bottles in stock.
- c. Get bin number for all red wines.
- d. Add three bottles to bin number 30.
- e. Remove all Chardonnay from stock.
- f. Add an entry for a new case (12 bottles) of Gary Farrell Merlot: bin number 55, year 1996, ready in 2001.

1.10 Suppose you have a classical music collection consisting of CDs and/or LPs and/or audio tapes, and you want to build a database that will let you find which recordings you have for a specific composer (e.g., Sibelius) or conductor (e.g., Simon Rattle) or soloist (e.g., Arthur Grumiaux) or work (e.g., Beethoven's Fifth) or orchestra (e.g., the NYPO) or kind of work (e.g., violin concerto) or chamber group (e.g., the Kronos Quartet). Draw an entity/relationship diagram like that of Fig. 1.6 for this database.

REFERENCES AND BIBLIOGRAPHY

1.1 E. F. Codd: "Data Models in Database Management," Proc. Workshop on Data Abstraction, Databases, and Conceptual Modelling, Pingree Park, Colo. (June 1980); *ACM SIGART Newsletter* No. 74 (January 1981); *ACM SIGMOD Record* 11, No. 2 (February 1981); *ACM SIGPLAN Notices* 16, No. 1 (January 1981).

Codd was the inventor of the relational model, which he first described in reference [5.1]. Reference [5.1], however, did not in fact define the term *data model* as such!—but the present (much later) paper does. It then addresses the question: What purposes are data models in general, and the relational model in particular, intended to serve? And it goes on to offer evidence to support the claim that, contrary to popular belief, the relational model was actually the first data model to be defined. (In other words, Codd has some claim to being the inventor of the data model concept in general, as well as of the relational data model in particular.)

1.2 Hugh Darwen: "What a Database Really Is: Predicates and Propositions," in C. J. Date, Hugh Darwen, and David McGoveran, *Relational Database Writings 1994–1997*. Reading, Mass.: Addison-Wesley (1998).

This paper gives an informal (but accurate) explanation of the idea, discussed briefly at the end of Section 1.3, that a database is best thought of as a collection of true propositions.

1.3 C. J. Date and P. Hopewell: "Storage Structures and Physical Data Independence," Proc. 1971 ACM SIGFIDET Workshop on Data Definition, Access, and Control, San Diego, California (November 1971).

1.4 C. J. Date and P. Hopewell: "File Definition and Logical Data Independence," Proc. 1971 ACM SIGFIDET Workshop on Data Definition, Access, and Control, San Diego, California (November 1971).

References [1.3–1.4] were the first papers to define and distinguish between physical and logical data independence.

1.5 C. J. Date: *Relational Database Writings 1991–1994*. Reading, Mass.: Addison-Wesley (1995).

ANSWERS TO SELECTED EXERCISES

1.3 Some disadvantages are as follows:

- Security might be compromised (without good controls);
- Integrity might be compromised (without good controls);
- Additional hardware might be required;
- Performance overhead might be significant;

- Successful operation is crucial (the enterprise might be highly vulnerable to failure);
- The system is likely to be complex (though such complexity should be concealed from the user).

1.6 a.

WINE	PRODUCER
Zinfandel	Rafanelli

b.

WINE	PRODUCER
Chardonnay	Buena Vista
Chardonnay	Geyser Peak
Joh. Riesling	Jekel
Fumé Blanc	Ch. St. Jean
Gewurztraminer	Ch. St. Jean

c.

BIN#	WINE	YEAR
6	Chardonnay	1996
22	Fumé Blanc	1996
52	Pinot Noir	1995

d.

WINE	BIN#	YEAR
Cab. Sauvignon	48	1993

1.7 We give a solution for part a. only: "Rafanelli is a producer of Zinfandel"—or, more precisely, "Some bin contains some bottles of Zinfandel that were produced by Rafanelli in some year, and they will be ready to drink in some year."

- 1.8 a. A row for bin number 80 is added to the CELLAR table.
 b. The rows for bin numbers 45, 48, 64, and 72 are deleted from the CELLAR table.
 c. The row for bin number 50 has the number of bottles set to 5.
 d. Same as c.

Incidentally, note how convenient it is to be able to refer to rows by their primary key value (the primary key for the CELLAR table is [BIN#]—see Chapter 8).

- 1.9 a.

```
SELECT BIN#, WINE, BOTTLES
FROM   CELLAR
WHERE  PRODUCER = 'Geyser Peak' ;
```
- b.

```
SELECT BIN#, WINE
FROM   CELLAR
WHERE  BOTTLES > 5 ;
```
- c.

```
SELECT BIN#
FROM   CELLAR
WHERE  WINE = 'Cab. Sauvignon'
OR     WINE = 'Pinot Noir'
OR     WINE = 'Zinfandel'
OR     WINE = 'Syrah'
OR     ..... ;
```

There is no shortcut answer to this question, because "color of wine" is not explicitly recorded in the database; thus, the DBMS does not know that (e.g.) Pinot Noir is red.

- d. UPDATE CELLAR
SET BOTTLES = BOTTLES + 3
WHERE BIN# = 30 ;
- e. DELETE
FROM CELLAR
WHERE WINE = 'Chardonnay' ;
- f. INSERT
INTO CELLAR (BIN#, WINE, PRODUCER, YEAR, BOTTLES, READY)
VALUES (55, 'Merlot', 'Gary Farrell', 1996, 12, 2001) ;

Database System Architecture

2.1 INTRODUCTION

We are now in a position to present an architecture for a database system. Our aim in presenting this architecture is to provide a framework on which subsequent chapters can build. Such a framework is useful for describing general database concepts and for explaining the structure of specific database systems—but we do not claim that every system can neatly be matched to this particular framework, nor do we mean to suggest that this particular architecture provides the only possible framework. In particular, “small” systems (see Chapter 1) will probably not support all aspects of the architecture. However, the architecture does seem to fit most systems reasonably well; moreover, it is basically identical to the architecture proposed by the ANSI/SPARC Study Group on Data Base Management Systems (the so-called ANSI/SPARC architecture—see references [2.1–2.2]). We choose not to follow the ANSI/SPARC terminology in every detail, however.

Note: This chapter resembles Chapter 1 insofar as, while an understanding of the material it contains is essential to a full appreciation of the structure and capabilities of a modern database system, it is again somewhat abstract and dry. As with Chapter 1, therefore, you might prefer just to give the material a “once over lightly” reading for now and come back to it later as it becomes more directly relevant to the topics at hand.

2.2 THE THREE LEVELS OF THE ARCHITECTURE

The ANSI/SPARC architecture is divided into three levels, known as the internal, conceptual, and external level, respectively (see Fig. 2.1). Broadly speaking:

- The **internal level** (also known as the *physical* level) is the one closest to physical storage—i.e., it is the one concerned with the way the data is physically stored;
- The **external level** (also known as the *user logical* level) is the one closest to the users—i.e., it is the one concerned with the way the data is seen by individual users; and
- The **conceptual level** (also known as the *community logical* level, or sometimes just the *logical* level, unqualified) is a level of indirection between the other two.

Observe that the external level is concerned with *individual* user perceptions, while the conceptual level is concerned with a *community* user perception. In other words, there will be

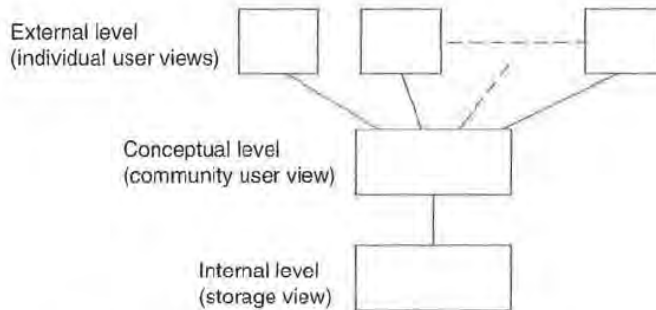


Fig. 2.1 The three levels of the architecture

many distinct "external views," each consisting of a more or less abstract representation of some portion of the total database, and there will be precisely one "conceptual view," consisting of a similarly abstract representation of the database in its entirety.* (Recall that most users will not be interested in the total database, but only in some restricted portion of it.) Likewise, there will be precisely one "internal view," representing the database as physically stored.

An example will make these ideas clearer. Fig. 2.2 shows the conceptual view, the corresponding internal view, and two corresponding external views (one for a PL/I user and one for a COBOL user), all for a simple personnel database. Of course, the example is completely hypothetical—it is not intended to resemble any real system—and many irrelevant details have deliberately been omitted. *Explanation:*

- At the conceptual level, the database contains information concerning an entity type called `EMPLOYEE`. Each individual employee has an `EMPLOYEE_NUMBER` (six characters), a `DEPARTMENT_NUMBER` (four characters), and a `SALARY` (five decimal digits).
- At the internal level, employees are represented by a stored record type called `STORED_EMP`, twenty bytes long. `STORED_EMP` contains four stored fields: a six-byte prefix (presumably containing control information such as flags or pointers), and three data fields corresponding to the three properties of employees. In addition, `STORED_EMP` records are indexed on the `EMP#` field by an index called `EMPX`, whose definition is not shown.
- The PL/I user has an external view of the database in which each employee is represented by a PL/I record containing two fields (department numbers are of no interest

*By *abstract* here, we merely mean that the representation in question involves constructs such as records and fields that are more user-oriented, as opposed to constructs such as bits and bytes that are more machine-oriented.

External (PL/I)	External (COBOL)
DCL 1 EMPP, 2 EMP# CHAR(6), 2 SAL FIXED BIN(31);	01 EMPC. 02 EMPNO PIC X(6). 02 DEPTNO PIC X(4).
Conceptual	
EMPLOYEE	
EMPLOYEE_NUMBER	CHARACTER (6)
DEPARTMENT_NUMBER	CHARACTER (4)
SALARY	NUMERIC (5)
Internal	
STORED EMP	BYTES=20
PREFIX	TYPE=BYTE(6),OFFSET=0
EMP#	TYPE=BYTE(6),OFFSET=6,INDEX=EMPX
DEPT#	TYPE=BYTE(4),OFFSET=12
PAY	TYPE=FULLWORD,OFFSET=16

Fig. 2.2 An example of the three levels

to this user and have therefore been omitted). The record type is defined by an ordinary PL/I structure declaration in accordance with the normal PL/I rules.

- Similarly, the COBOL user has an external view in which each employee is represented by a COBOL record containing, again, two fields (this time, salaries have been omitted). The record type is defined by an ordinary COBOL record description in accordance with the normal COBOL rules.

Notice that corresponding data items can have different names at different points. For example, the employee number is called EMP# in the PL/I external view, EMPNO in the COBOL external view, EMPLOYEE_NUMBER in the conceptual view, and EMP# again in the internal view. Of course, the system must be aware of the correspondences; for example, it must be told that the COBOL field EMPNO is derived from the conceptual field EMPLOYEE_NUMBER, which in turn is derived from the stored field EMP# at the internal level. Such correspondences, or **mappings**, are not explicitly shown in Fig. 2.2; refer to Section 2.6 for further discussion.

Now, it makes little difference for the purposes of the present chapter whether the system under consideration is relational or otherwise. However, it might be helpful to indicate briefly how the three levels of the architecture are typically realized in a relational system specifically:

- First, the conceptual level in such a system will definitely be relational, in the sense that the objects visible at that level will be relational tables and the operators will be relational operators (including in particular the *restrict* and *project* operators discussed briefly in Chapter 1).
- Second, a given external view will typically be relational also, or something very close to it; for example, the PL/I and COBOL record declarations of Fig. 2.2 might loosely be regarded as, respectively, the PL/I and COBOL analogs of the declaration of a relational table in a relational system.

Note: We should mention in passing that the term “external view” (usually abbreviated to just “view”) unfortunately has a rather specific meaning in relational contexts that is *not* identical to the meaning ascribed to it in this chapter. See Chapters 3 and 9 for an explanation and discussion of the relational meaning.

- Third, the internal level will *not* be relational, because the objects at that level will not be just (stored) relational tables—instead, they will be the same kinds of objects found at the internal level of any other kind of system (stored records, pointers, indexes, hashes, etc.). In fact, the relational model as such has **nothing whatsoever to say** about the internal level; it is, to repeat from Chapter 1, concerned with how the database looks to the *user*.

We now proceed to discuss the three levels of the architecture in considerably more detail, starting with the external level. Throughout our discussions we will be making repeated references to Fig. 2.3, which shows the major components of the architecture and their interrelationships.

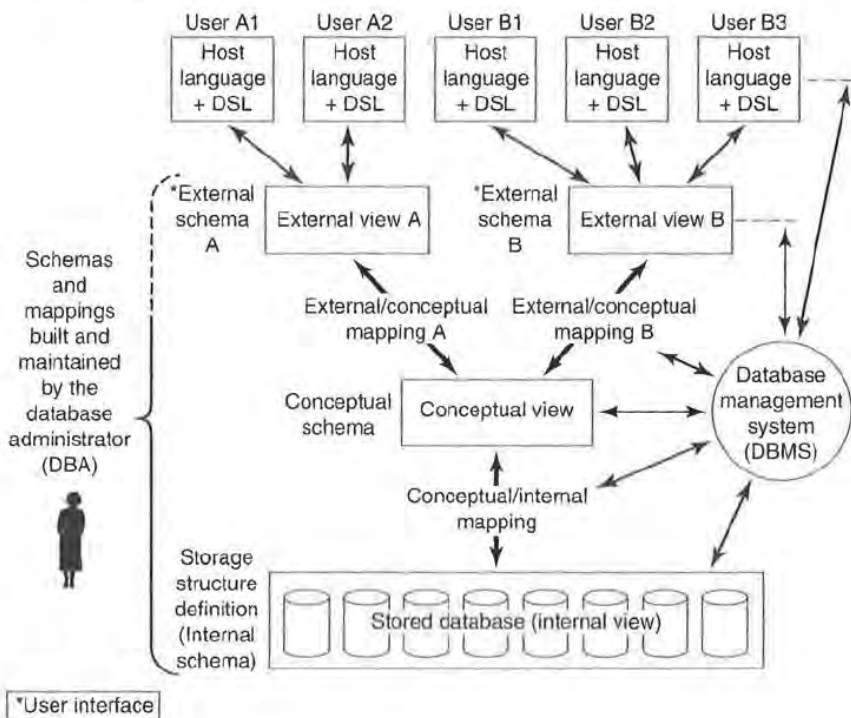


Fig. 2.3 Detailed system architecture

2.3 THE EXTERNAL LEVEL

The external level is the individual user level. As explained in Chapter 1, a given user can be either an application programmer or an end user of any degree of sophistication. (The DBA is an important special case; unlike other users, however, the DBA will need to be interested in the conceptual and internal levels also. See the next two sections.)

Each user has a **language** at his or her disposal:

- For the application programmer, that language will be either a conventional programming language (e.g., PL/I, C++, Java) or else a proprietary language that is specific to the system in question. Such proprietary languages are often called “fourth generation” languages (4GLs), on the—fuzzy!—grounds that (a) machine code, assembler language, and languages such as PL/I can be regarded as three earlier language “generations,” and (b) the proprietary languages represent the same kind of improvement over “third generation” languages (3GLs) as those languages did over assembler language and assembler language did over machine code.
- For the end user, the language will be either a query language or some special-purpose language, perhaps forms- or menu-driven, tailored to that user’s requirements and supported by some online application program (as explained in Chapter 1).

For our purposes, the important thing about all such languages is that they will include a **data sublanguage**—i.e., a subset of the total language that is concerned specifically with database objects and operations. The data sublanguage (abbreviated DSL in Fig. 2.3) is said to be **embedded** within the corresponding **host language**. The host language is responsible for providing various nondatabase facilities, such as local variables, computational operations, branching logic, and so on. A given system might support any number of host languages and any number of data sublanguages; however, one particular data sublanguage that is supported by almost all current systems is the language SQL discussed briefly in Chapter 1. Most such systems allow SQL to be used both *interactively* as a standalone query language and also *embedded* in other languages such as PL/I or Java (see Chapter 4 for further discussion).

Now, although it is convenient for architectural purposes to distinguish between the data sublanguage and its containing host language, the two might in fact *not* be distinct so far as the user is concerned; indeed, it is probably preferable from the user’s perspective if they are not. If they are not distinct, or if they can be distinguished only with difficulty, we say they are **tightly coupled**. If they are clearly and easily separable, we say they are **loosely coupled**. While a few commercial systems (especially object systems—see Chapter 24) do support tight coupling, most do not; SQL systems in particular typically support loose coupling only. (Tight coupling provides a more uniform set of facilities for the user but obviously involves more effort on the part of the system developers, a fact that presumably accounts for the *status quo*.)

In principle, any given data sublanguage is really a combination of at least two subordinate languages—a **data definition language** (DDL), which supports the definition or

declaration of database objects, and a **data manipulation language** (DML), which supports the manipulation or processing of such objects. For example, consider the PL/I user of Fig. 2.2 in Section 2.2. The data sublanguage for that user consists of those PL/I features that are used to communicate with the DBMS:

- The DDL portion consists of those declarative constructs of PL/I that are needed to declare database objects—the DECLARE (DCL) statement itself, certain PL/I data types, possibly special extensions to PL/I to support new objects that are not handled by existing PL/I.
- The DML portion consists of those executable statements of PL/I that transfer information to and from the database—again, possibly including special new statements.

Note: In the interests of accuracy, we should say that PL/I in fact does not include any specific database features at all at the time of writing. The “DML” statements in particular are typically just PL/I CALL statements that invoke the DBMS (though those calls might be syntactically disguised in some way to make them a little more user-friendly—see the discussion of embedded SQL in Chapter 4).

To return to the architecture: We have already indicated that an individual user will generally be interested only in some portion of the total database; moreover, that user’s view of that portion will generally be somewhat abstract when compared with the way the data is physically stored. The ANSI/SPARC term for an individual user’s view is an **external view**. An external view is thus the content of the database as seen by some particular user (that is, to that user the external view *is* the database). For example, a user from the Personnel Department might regard the database as a collection of department and employee record occurrences, and might be quite unaware of the supplier and part record occurrences seen by users in the Purchasing Department.

In general, then, an external view consists of many occurrences of each of many types of **external record** (not necessarily the same thing as a stored record).^{*} The user’s data sublanguage is defined in terms of external records; for example, a DML *retrieve* operation will retrieve external record occurrences, not stored record occurrences. *Note:* We can now see that the term “logical record” used a couple of times in Chapter 1 actually referred to an external record. From this point forward, in fact, we will generally avoid the term “logical record.”

Each external view is defined by means of an **external schema**, which consists basically of definitions of each of the various external record types in that external view (again, refer back to Fig. 2.2 for a couple of simple examples). The external schema is written using the DDL portion of the user’s data sublanguage. (That DDL is therefore sometimes referred to as an *external DDL*.) For example, the employee external record type might be defined as a six-character employee number field plus a five-digit (decimal) salary field, and so on.

^{*}We are assuming here that all information is represented at the external level in the form of records. However, some systems allow information to be represented in other ways instead of or as well as records. For a system using such alternative methods, the definitions and explanations given in this section will require suitable modification. Analogous remarks apply to the conceptual and internal levels also, *q.v.* Detailed consideration of such matters is beyond the scope of this early part of the book; see Chapters 13 (especially the “References and Bibliography” section) and 24 for further discussion.

In addition, there must be a definition of the *mapping* between the external schema and the underlying *conceptual* schema (see the next section). We will consider that mapping later, in Section 2.6.

2.4 THE CONCEPTUAL LEVEL

The **conceptual view** is a representation of the entire information content of the database, again (as with an external view) in a form that is somewhat abstract in comparison with the way in which the data is physically stored. It will also be quite different, in general, from the way in which the data is viewed by any particular user. Broadly speaking, the conceptual view is intended to be a view of the data “as it really is,” rather than as users are forced to see it by the limitations of (for example) the particular language or hardware they might be using.

The conceptual view consists of many occurrences of each of many types of **conceptual record**. For example, it might consist of a collection of department record occurrences plus a collection of employee record occurrences plus a collection of supplier record occurrences plus a collection of part record occurrences (etc., etc.). A conceptual record is not necessarily the same as either an external record on the one hand, or a stored record on the other.

The conceptual view is defined by means of the **conceptual schema**, which includes definitions of each of the various conceptual record types (again, refer to Fig. 2.2 for a simple example). The conceptual schema is written using another data definition language, the *conceptual DDL*. If physical data independence is to be achieved, then those conceptual DDL definitions must not involve any considerations of physical representation or access technique at all—they must be definitions of information content *only*. Thus, there must be no reference in the conceptual schema to stored field representation, stored record sequence, indexes, hashing schemes, pointers, or any other storage and access details. If the conceptual schema is made truly data-independent in this way, then the external schemas, which are defined in terms of the conceptual schema (see Section 2.6), will *a fortiori* be data-independent too.

The conceptual view, then, is a view of the total database content, and the conceptual schema is a definition of that view. However, it would be misleading to suggest that the conceptual schema is nothing more than a set of definitions much like the simple record definitions found in (e.g.) a COBOL program today. The definitions in the conceptual schema are intended to include a great many additional features, such as the security and integrity constraints mentioned in Chapter 1. Some authorities would go so far as to suggest that the ultimate objective of the conceptual schema is to describe the complete enterprise—not just its data *per se*, but also how that data is used; how it flows from point to point within the enterprise, what it is used for at each point, what audit or other controls are to be applied at each point, and so on [2.3]. It must be emphasized, however, that no system today actually supports a conceptual schema of anything approaching this degree of comprehensiveness; in most existing systems, the “conceptual schema” is really little more than a simple union of all of the individual external schemas, plus certain security and integrity constraints. But it is certainly possible that systems of the future will be much more sophisticated in their support of the conceptual level.

2.5 THE INTERNAL LEVEL

The third level of the architecture is the internal level. The **internal view** is a low-level representation of the entire database; it consists of many occurrences of each of many types of **internal record**. “Internal record” is the ANSI/SPARC term for the construct that we have been calling a *stored* record (and we will continue to use this latter term). The internal view is thus still at one remove from the physical level, since it does not deal in terms of *physical* records—also called **blocks** or **pages**—nor with any device-specific considerations such as cylinder or track sizes. In other words, the internal view effectively assumes an infinite linear address space; details of how that address space is mapped to physical storage are highly system-specific and are deliberately omitted from the general architecture. *Note:* The block, or page, is **the unit of I/O**—i.e., it is the amount of data transferred between secondary storage and main memory in a single I/O operation. Typical page sizes are 1K, 2K, or 4K bytes ($K = 1024$).

The internal view is described by means of the **internal schema**, which not only defines the various stored record types but also specifies what indexes exist, how stored fields are represented, what physical sequence the stored records are in, and so on (once again, see Fig. 2.2 for a simple example). The internal schema is written using yet another data definition language—the *internal DDL*. *Note:* In this book we will normally use the more intuitive terms “storage structure” or “stored database” in place of “internal view” and “storage structure definition” in place of “internal schema.”

In closing, we remark that, in certain exceptional situations, application programs—in particular, applications of a *utility* nature (see Section 2.11)—might be permitted to operate directly at the internal level rather than at the external level. Needless to say, the practice is not recommended; it represents a security risk (since the security constraints are bypassed) and an integrity risk (since the integrity constraints are bypassed likewise), and the program will be data-dependent to boot; but sometimes it might be the only way to obtain the required functionality or performance—just as the user in a high-level programming language system might occasionally need to descend to assembler language in order to satisfy certain functionality or performance objectives today.

2.6 MAPPINGS

In addition to the three levels *per se*, the architecture of Fig. 2.3 involves certain **mappings**—one conceptual/internal mapping and several external/conceptual mappings, in general:

- The *conceptual/internal* mapping defines the correspondence between the conceptual view and the stored database; it specifies how conceptual records and fields are represented at the internal level. If the structure of the stored database is changed—i.e., if a change is made to the storage structure definition—then the conceptual/internal mapping must be changed accordingly, so that the conceptual schema can remain invari-

ant. (It is the responsibility of the DBA to manage such changes, of course.) In other words, the effects of such changes must be isolated below the conceptual level, in order to preserve physical data independence.

- An *external/conceptual* mapping defines the correspondence between a particular external view and the conceptual view. In general, the differences that can exist between these two levels are analogous to those that can exist between the conceptual view and the stored database. For example, fields can have different data types; field and record names can be changed; several conceptual fields can be combined into a single (virtual) external field; and so on. Any number of external views can exist at the same time; any number of users can share a given external view; different external views can overlap.

Note: It should be clear that, just as the conceptual/internal mapping is the key to physical data independence, so the external/conceptual mappings are the key to **logical** data independence. As we saw in Chapter 1, a system provides *physical* data independence [1.3] if users and user programs are immune to changes in the physical structure of the stored database. Analogously, a system provides *logical* data independence [1.4] if users and user programs are also immune to changes in the *logical* structure of the database (meaning changes at the conceptual or “community logical” level). We will have more to say on this important issue in Chapters 3 and 9.

Incidentally, most systems permit the definition of certain external views to be expressed in terms of others (in effect, via *external/external* mappings), rather than always requiring an explicit definition of the mapping to the conceptual level—a useful feature if several external views are rather similar to one another. Relational systems in particular provide such a capability.

2.7 THE DATABASE ADMINISTRATOR

As explained in Chapter 1, the *data* administrator (DA) is the person who makes the strategic and policy decisions regarding the data of the enterprise, and the *database* administrator (DBA) is the person who provides the necessary technical support for implementing those decisions. Thus, the DBA is responsible for the overall control of the system at a technical level. We can now describe some of the tasks of the DBA in a little more detail. In general, those tasks will include at least all of the following:

- *Defining the conceptual schema*

It is the data administrator’s job to decide exactly what information is to be held in the database—in other words, to identify the entities of interest to the enterprise and to identify the information to be recorded about those entities. This process is usually referred to as **logical**—sometimes *conceptual*—**database design**. Once the data administrator has thus decided the content of the database at an abstract level, the DBA will then create the corresponding conceptual schema, using the conceptual DDL. The object (compiled) form of that schema will be used by the DBMS in responding to access

requests. The source (uncompiled) form will act as a reference document for the users of the system.

Note: In practice, matters might not be as clearcut as the foregoing remarks suggest. In some cases, the data administrator might create the conceptual schema directly. In others, the DBA might do the logical design.

■ *Defining the internal schema*

The DBA must also decide how the data is to be represented in the stored database. This process is usually referred to as **physical** database design.* Having done the physical design, the DBA must then create the corresponding storage structure definition (i.e., the internal schema), using the internal DDL. In addition, he or she must also define the associated conceptual/internal mapping. In practice, either the conceptual DDL or the internal DDL—most likely the former—will probably include the means for defining that mapping, but the two functions (creating the schema, defining the mapping) should be clearly separable. Like the conceptual schema, the internal schema and corresponding mapping will exist in both source and object form.

■ *Liaising with users*

It is the business of the DBA to liaise with users to ensure that the data they need is available and to write (or help the users write) the necessary external schemas, using the applicable external DDL. (As already mentioned, a given system might support several distinct external DDLs.) In addition, the corresponding external/conceptual mappings must also be defined. In practice, the external DDL will probably include the means for specifying those mappings, but once again the schemas and the mappings should be clearly separable. Each external schema and corresponding mapping will exist in both source and object form.

Other aspects of the user liaison function include consulting on application design; providing technical education; assisting with problem determination and resolution; and similar professional services.

■ *Defining security and integrity constraints*

As already discussed, security and integrity constraints can be regarded as part of the conceptual schema. The conceptual DDL must include facilities for specifying such constraints.

■ *Defining dump and reload policies*

Once an enterprise is committed to a database system, it becomes critically dependent on the successful operation of that system. In the event of damage to any portion of the database—caused by human error, say, or a failure in the hardware or operating system—it is essential to be able to repair the data concerned with the minimum of delay and with as little effect as possible on the rest of the system. For example, the availability of data that has *not* been damaged should ideally not be affected. The DBA must

*Observe the sequence: Decide what data you want first, then decide how to represent it in storage. Physical design should always be done *after* logical design.

define and implement an appropriate damage control scheme, typically involving (a) periodic unloading or “dumping” of the database to backup storage and (b) reloading the database when necessary from the most recent dump.

Incidentally, the need for quick data repair is one reason why it might be a good idea to spread the total data collection across several databases, instead of keeping it all in one place; the individual database might very well form the unit for dump and reload purposes. In this connection, note that *terabyte systems**—i.e., commercial database installations that store well over a trillion bytes of data, loosely speaking—already exist, and systems of the future are predicted to be much larger. It goes without saying that such *VLDB* (“very large database”) systems require very careful and sophisticated administration, especially if there is a requirement for continuous availability (which there usually is). Nevertheless, we will continue to talk as if there were in fact just a single database, for simplicity.

■ *Monitoring performance and responding to changing requirements*

As indicated in Chapter 1, the DBA is responsible for organizing the system in such a way as to get the performance that is “best for the enterprise,” and for making the appropriate adjustments—i.e., **tuning**—as requirements change. For example, it might be necessary to **reorganize** the stored database from time to time to ensure that performance levels remain acceptable. As already mentioned, any change to the physical storage (internal) level of the system must be accompanied by a corresponding change to the definition of the conceptual/internal mapping, so that the conceptual schema can remain constant.

Of course, the foregoing is not an exhaustive list—it is merely intended to give some idea of the extent and nature of the DBA’s responsibilities.

2.8 THE DATABASE MANAGEMENT SYSTEM

The **database management system** (DBMS) is the software that handles all access to the database. Conceptually, what happens is the following (refer to Fig. 2.3 once again):

1. A user issues an access request, using some particular data sublanguage (typically SQL).
2. The DBMS intercepts that request and analyzes it.
3. The DBMS inspects, in turn, (the object versions of) the external schema for that user, the corresponding external/conceptual mapping, the conceptual schema, the conceptual/internal mapping, and the storage structure definition.
4. The DBMS executes the necessary operations on the stored database.

*1024 bytes = one kilobyte (KB); 1024KB = one megabyte (MB); 1024MB = one gigabyte (GB); 1024GB = one terabyte (TB); 1024TB = one petabyte (PB); 1024PB = one exabyte (EB). Note that a gigabyte is a billion bytes, loosely speaking (the abbreviation BB is sometimes used in place of GB). Incidentally—and contrary to popular belief—*gigabyte* is pronounced with a soft initial g (“jigabyte”).

By way of an example, consider what is involved in the retrieval of a particular external record occurrence. In general, fields will be required from several conceptual record occurrences, and each conceptual record occurrence in turn will require fields from several stored record occurrences. Conceptually, then, the DBMS must first retrieve all required stored record occurrences, then construct the required conceptual record occurrences, and then construct the required external record occurrence. At each stage, data type or other conversions might be necessary.

Of course, the foregoing description is very much simplified; in particular, it implies that the entire process is interpretive, inasmuch as it suggests that the processes of analyzing the request, inspecting the various schemas, etc., are all done at run time. Interpretation, in turn, often implies poor performance, because of the run-time overhead. In practice, however, it might be possible for access requests to be *compiled* prior to run time (in particular, several of today's SQL products do this—see the annotation to references [4.12] and [4.26] in Chapter 4).

Let us now examine the functions of the DBMS in a little more detail. Those functions will include support for at least all of the following (refer to Fig. 2.4):

■ *Data definition*

The DBMS must be able to accept data definitions (external schemas, the conceptual schema, the internal schema, and all associated mappings) in source form and convert them to the appropriate object form. In other words, the DBMS must include **DDL processor** or **DDL compiler** components for each of the various data definition languages (DDLs). The DBMS must also “understand” the DDL definitions, in the sense that, for example, it “understands” that **EMPLOYEE** external records include a **SALARY** field; it must then be able to use this knowledge in analyzing and responding to data manipulation requests (e.g., “Get all employees with salary < \$50,000”).

■ *Data manipulation*

The DBMS must be able to handle requests to retrieve, update, or delete existing data in the database or to add new data to the database. In other words, the DBMS must include a **DML processor** or **DML compiler** component to deal with the data manipulation language (DML).

■ In general, DML requests can be “planned” or “unplanned”:

- a. A **planned** request is one for which the need was foreseen well in advance of the time at which the request is executed. The DBA will probably have tuned the physical database design in such a way as to guarantee good performance for planned requests.
- b. An **unplanned** request, by contrast, is an *ad hoc* query, i.e., a request for which the need was not seen in advance, but instead arose in a spur-of-the-moment fashion. The physical database design might or might not be ideally suited for the specific request under consideration.

To use the terminology introduced in Chapter 1 (Section 1.3), planned requests are characteristic of “operational” or “production” applications, while unplanned requests are characteristic of “decision support” applications. Furthermore, planned requests

under the control of the **run-time manager**. *Note:* In practice, the run-time manager will probably invoke some kind of *file manager* to access the stored data. File managers are discussed briefly at the end of the present section.

- *Data security and integrity*

The DBMS must monitor user requests and reject any attempts to violate the security and integrity constraints defined by the DBA (see the previous section). These tasks can be carried out at compile time or run time or some mixture of the two.

- *Data recovery and concurrency*

The DBMS—or, more likely, some other related software component, usually called the **transaction manager** or **transaction processing monitor** (TP monitor)—must enforce certain recovery and concurrency controls. Details of these aspects of the system are beyond the scope of this chapter; see Part IV of this book for an indepth discussion. *Note:* The transaction manager is not shown in Fig. 2.4 because it is usually not part of the DBMS *per se*.

- *Data dictionary*

The DBMS must provide a **data dictionary** function. The data dictionary can be regarded as a database in its own right (but a system database rather than a user database). The dictionary contains “data about the data” (sometimes called *metadata* or *descriptors*)—that is, *definitions* of other objects in the system, instead of just “raw data.” In particular, all of the various schemas and mappings (external, conceptual, etc.) and all of the various security and integrity constraints will be stored, in both source and object form, in the dictionary. A comprehensive dictionary will also include much additional information, showing, for instance, which programs use which parts of the database, which users require which reports, and so on. The dictionary might even—in fact, probably should—be integrated into the database it defines and thus include its own definition. Certainly it should be possible to query the dictionary just like any other database, so that, for example, it is possible to tell which programs and/or users are likely to be affected by some proposed change to the system. See Chapter 3 for further discussion.

Note: We are touching here on an area in which there is much terminological confusion. Some people would refer to what we are calling the dictionary as a *directory* or a *catalog*—with the implication that directories and catalogs are somehow inferior to a true dictionary—and would reserve the term *dictionary* to refer to a specific (important) kind of application development tool. Other terms that are also sometimes used to refer to this latter kind of object are *data repository* (see Chapter 13) and *data encyclopedia*.

- *Performance*

It goes without saying that the DBMS should perform all of the tasks identified above as efficiently as possible.

We can summarize all of the foregoing by saying that the overall purpose of the DBMS is to provide the **user interface** to the database system. The user interface can be defined

as a boundary in the system below which everything is invisible to the user. By definition, therefore, the user interface is at the *external* level. However, as we will see in Chapter 9, there are some situations in which the external view is unlikely to differ very significantly from the relevant portion of the underlying conceptual view, at least in today's commercial SQL products.

We conclude this section by briefly contrasting database management systems as discussed above with **file** management systems (*file managers* or *file servers* for short). Basically, the file manager is that component of the underlying operating system that manages stored files; loosely speaking, therefore, it is "closer to the disk" than the DBMS is. (In fact, the DBMS is typically built on top of some kind of file manager.) Thus, the user of a file management system will be able to create and destroy stored files and perform simple retrieval and update operations on stored records in such files. In contrast to the DBMS, however:

- File managers are not aware of the internal structure of stored records and hence cannot handle requests that rely on a knowledge of that structure.
- They typically provide little or no support for security and integrity constraints.
- They typically provide little or no support for recovery and concurrency controls.
- There is no true data dictionary concept at the file manager level.
- They provide much less data independence than the DBMS does.
- Files are typically not "integrated" or "shared" in the same sense that the database is (they are usually private to some particular user or application).

2.9 THE DATA COMMUNICATIONS MANAGER

In this section, we briefly consider the topic of **data communications**. Database requests from an end user are actually transmitted from the user's workstation—which might be physically remote from the database system itself—to some online application (builtin or otherwise), and thence to the DBMS, in the form of *communication messages*. Likewise, responses back from the DBMS and online application to the user's workstation are also transmitted in the form of such messages. All such message transmissions take place under the control of another software component, the **data communications manager** (DC manager).

The DC manager is not part of the DBMS but is an autonomous system in its own right. However, since it is clearly required to work in harmony with the DBMS, the two are sometimes regarded as equal partners in a higher-level cooperative venture called the **database/data-communications system** (DB/DC system), in which the DBMS looks after the database and the DC manager handles all messages to and from the DBMS, or more accurately to and from applications that use the DBMS. In this book, however, we will have comparatively little to say about message handling as such (it is a large subject in its own right). Section 2.12 does briefly discuss the question of communication *between distinct systems* (i.e., between distinct machines in a communications network, such as the Internet), but that is really a separate topic.

2.10 CLIENT/SERVER ARCHITECTURE

So far in this chapter we have been discussing database systems from the point of view of the so-called ANSI/SPARC architecture. In particular, we gave a simplified picture of that architecture in Fig. 2.3. In this section we take a look at database systems from a slightly different perspective. The overall purpose of such systems, of course, is to support the development and execution of database applications. From a high-level point of view, therefore, a database system can be regarded as having a very simple two-part structure, consisting of a **server** (also called the *backend*) and a set of **clients** (also called *frontends*). Refer to Fig. 2.5. *Explanation:*

- The *server* is just the DBMS itself. It supports all of the basic DBMS functions discussed in Section 2.8—data definition, data manipulation, data security and integrity, and so on. In particular, it provides all of the external, conceptual, and internal level support discussed in Sections 2.3–2.6. Thus, “server” in this context is just another name for the DBMS.
- The *clients* are the various applications that run on top of the DBMS—both user-written applications and builtin applications, i.e., applications provided by the DBMS vendor or by some third party. As far as the server is concerned, of course, there is no difference between user-written and builtin applications—they all use the same interface to the server, namely the external-level interface discussed in Section 2.3.

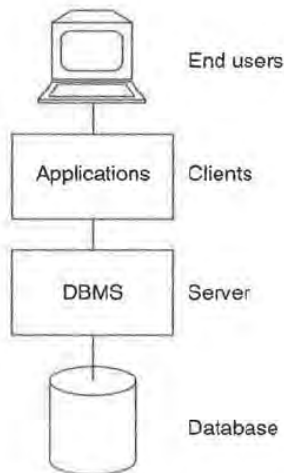


Fig. 2.5 Client/server architecture

Note: Certain special “utility” applications might constitute an exception to the foregoing, inasmuch as they might sometimes need to operate directly at the *internal* level of the system (as mentioned in Section 2.5). Such utilities are best regarded as integral components of the DBMS, rather than as applications in the usual sense. They are discussed in more detail in the next section.

We elaborate briefly on the question of user-written vs. vendor-provided applications:

- *User-written applications* are basically regular application programs, written (typically) either in a conventional 3GL such as C or COBOL or in some proprietary 4GL—though in both cases the language needs to be coupled somehow with an appropriate data sublanguage, as explained in Section 2.3.
- *Vendor-provided applications* (often called **tools**) are applications whose basic purpose is to assist in the creation and execution of other applications! The applications that are created are applications that are tailored to some specific task (they might not look much like applications in the conventional sense; indeed, the whole point of the tools is to allow users, especially end users, to create applications *without* having to write programs in a conventional programming language). For example, one of the vendor-provided tools will be a *report writer*, whose purpose is to allow end users to obtain formatted reports from the system on request. Any given report request can be regarded as a small application program, written in a very high-level (and special purpose) *report writer language*.

Vendor-provided tools can be divided into several more or less distinct classes:

- a. Query language processors;
- b. Report writers;
- c. Business graphics subsystems;
- d. Spreadsheets;
- e. Natural language processors;
- f. Statistical packages;
- g. Copy management or “data extract” tools;
- h. Application generators (including 4GL processors);
- i. Other application development tools, including computer-aided software engineering (CASE) products;

and many others. Details of most such tools are beyond the scope of this book; however, we remark that since (as stated above) the whole point of a database system is to support the creation and execution of applications, the quality of the available tools is, or should be, a major factor in “the database decision” (i.e., the process of choosing the right database product). In other words, the DBMS *per se* is not the only factor that needs to be taken into account, nor even necessarily the most significant factor.

We close this section with a forward reference. Since the overall system can be so neatly divided into two parts, server and clients, the possibility arises of running the two on

different machines. In other words, the potential exists for **distributed processing**. Distributed processing means that distinct machines can be connected together into some kind of communications network in such a way that a single data-processing task can be spread across several machines in the network. In fact, so attractive is this possibility—for a variety of reasons, mainly economic—that the term “client/server” has come to apply almost exclusively to the case where client and server are indeed on different machines. We will discuss distributed processing in more detail in Section 2.12.

2.11 UTILITIES

Utilities are programs designed to help the DBA with various administration tasks. As mentioned in the previous section, some utility programs operate at the external level of the system, and thus are effectively nothing more than special-purpose applications; some might not even be provided by the DBMS vendor, but rather by some third party. Other utilities, however, operate directly at the internal level (in other words, they are really part of the server), and hence must be provided by the DBMS vendor.

Here are some examples of the kind of utilities that are typically needed in practice:

- **Load** routines, to create the initial version of the database from one or more operating system files;
- **Unload/reload** routines, to unload the database, or portions thereof, to backup storage and to reload data from such backup copies (of course, the “reload utility” is basically identical to the load utility just mentioned);
- **Reorganization** routines, to rearrange the data in the stored database for various reasons (usually having to do with performance)—e.g., to cluster data together in some particular way on the disk, or to reclaim space occupied by data that has become obsolete;
- **Statistical** routines, to compute various performance statistics such as file sizes or value distributions or I/O counts, etc.;
- **Analysis** routines, to analyze the statistics just mentioned;

and so on. Of course, this list represents just a small sample of the range of functions that utilities typically provide; a wealth of other possibilities exist.

2.12 DISTRIBUTED PROCESSING

To repeat from Section 2.10, the term “distributed processing” means that distinct machines can be connected together into a communications network such as the Internet, such that a single data-processing task can span several machines in the network. (The term “parallel processing” is also sometimes used with essentially the same meaning, except that the distinct machines tend to be physically close together in a “parallel” system and need not be

so in a “distributed” system—e.g., they might be geographically dispersed in the latter case.) Communication among the various machines is handled by some kind of network management software—possibly an extension of the DC manager discussed in Section 2.9, more likely a separate software component.

Many levels or varieties of distributed processing are possible. To repeat from Section 2.10, one simple case involves running the DBMS backend (the server) on one machine and the application frontends (the clients) on another. Refer to Fig. 2.6.

As mentioned at the end of Section 2.10, “client/server,” though strictly speaking a purely architectural term, has come to be almost synonymous with the arrangement illustrated in Fig. 2.6, in which client and server run on different machines. Indeed, there are many arguments in favor of such a scheme:

- The first is basically just the usual parallel-processing argument: namely, many processing units are now being applied to the overall task, and server (database) and client (application) processing are being done in parallel. Response time and throughput should thus be improved.
- Furthermore, the server machine might be a custom-built machine that is tailored to the DBMS function (a “database machine”) and might thus provide better DBMS performance.

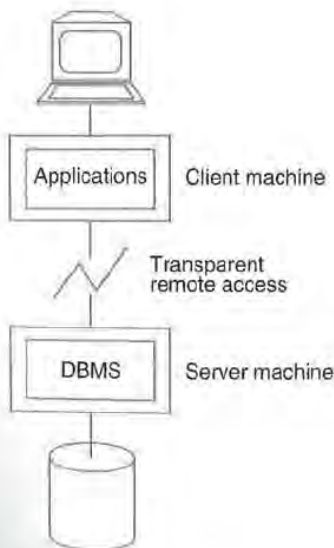


Fig. 2.6 Client(s) and server running on different machines

- Likewise, the client machine might be a personal workstation, tailored to the needs of the end user and thus able to provide better interfaces, high availability, faster responses, and overall improved ease of use to the user.
- Several different client machines might be able (in fact, typically will be able) to access the same server machine. Thus, a single database might be shared across several distinct client systems (see Fig. 2.7).

In addition to the foregoing arguments, there is also the point that running the client(s) and the server on separate machines matches the way enterprises actually operate. It is quite common for a single enterprise—a bank, for example—to operate many computers, such that the data for one portion of the enterprise is stored on one computer and the data for another portion is stored on another. It is also quite common for users on one computer to need at least occasional access to data stored on another. To pursue the banking example for a moment, it is very likely that users at one branch office will occasionally need access to data stored at another. Note, therefore, that the client machines might have stored data of their own, and the server machine might have applications of its own. In general, therefore, each machine will act as a server for some users and a client for others (see Fig. 2.8); in other words, each machine will support *an entire database system*, in the sense of earlier sections of this chapter.

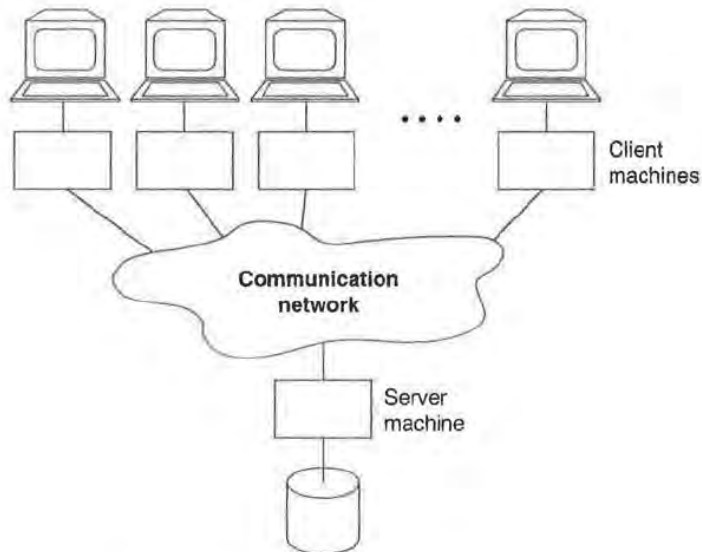


Fig. 2.7 One server machine, many client machines

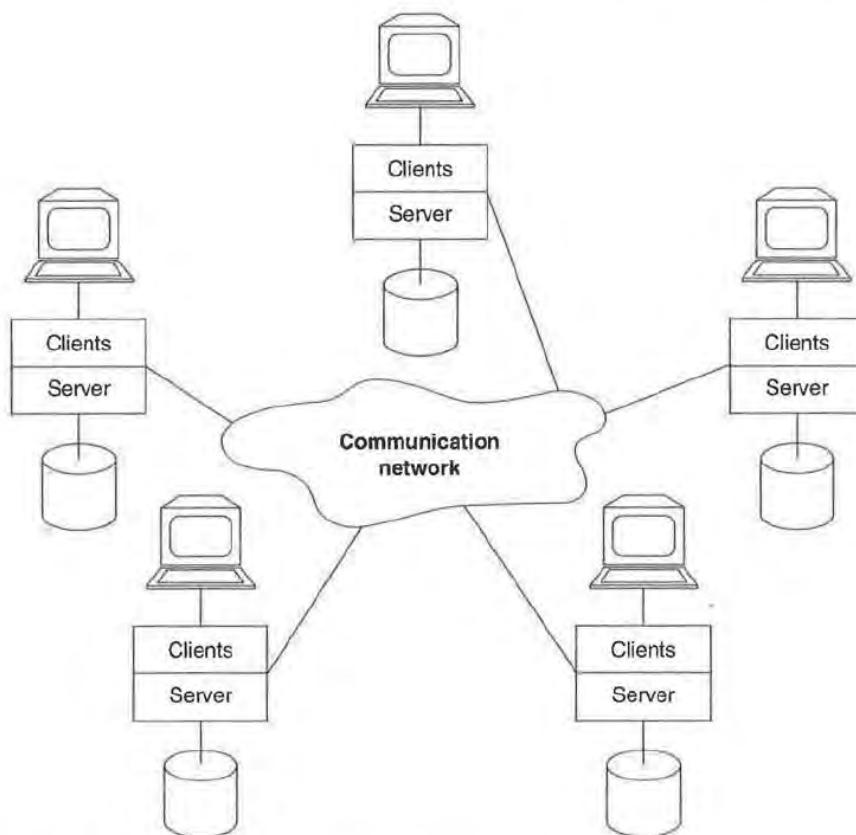


Fig. 2.8 Each machine runs both client(s) and server

The final point is that a single client machine might be able to access several different server machines (the converse of the case illustrated in Fig. 2.7). This capability is desirable because, as mentioned above, enterprises do typically operate in such a manner that the totality of their data is not stored on one single machine but rather is spread across many distinct machines, and applications will sometimes need the ability to access data from more than one machine. Such access can basically be provided in two different ways:

- A given client might be able to access any number of servers, but only one at a time (i.e., each individual database request must be directed to just one server). In such a system it is not possible, within a single request, to combine data from two or more

different servers. Furthermore, the user in such a system has to know which particular machine holds which pieces of data.

- The client might be able to access many servers simultaneously (i.e., a single database request might be able to combine data from several servers). In this case, the servers look to the client—from a logical point of view—as if they were really a single server, and the user does not have to know which machines hold which pieces of data.

This latter case constitutes what is usually called a **distributed database system**. Distributed database is a big topic in its own right; carried to its logical conclusion, full distributed database support implies that a single application should be able to operate “transparently” on data that is spread across a variety of different databases, managed by a variety of different DBMSs, running on a variety of different machines, supported by a variety of different operating systems, and connected together by a variety of different communication networks—where “transparently” means that the application operates from a logical point of view as if the data were all managed by a single DBMS running on a single machine. Such a capability might sound like a pretty tall order!—but it is highly desirable from a practical perspective, and vendors are working hard to make such systems a reality. We will discuss such systems in detail in Chapter 20.

2.13 SUMMARY

In this chapter we have looked at database systems from an architectural point of view. First, we described the **ANSI/SPARC architecture**, which divides a database system into three levels, as follows: The **internal** level is the one closest to physical storage (i.e., it is the one concerned with the way the data is physically stored); the **external** level is the one closest to the users (i.e., it is the one concerned with the way the data is viewed by individual users); and the **conceptual** level is a level of indirection between the other two (it provides a *community view* of the data). The data as perceived at each level is described by a **schema** (or several schemas, in the case of the external level). **Mappings** define the correspondence between (a) a given external schema and the conceptual schema, and (b) the conceptual schema and the internal schema. Those mappings are the key to the provision of **logical** and **physical data independence**, respectively.

Users—i.e., end users and application programmers, both of whom operate at the external level—interact with the data by means of a **data sublanguage**, which divides into at least two components, a **data definition language** (DDL) and a **data manipulation language** (DML). The data sublanguage is embedded in a **host language**. *Note:* The boundaries between the host language and the data sublanguage and between the DDL and the DML are primarily conceptual in nature; ideally they should be “transparent to the user.”

We also took a closer look at the functions of the **DBA** and the **DBMS**. Among other things, the DBA is responsible for creating the internal schema (**physical database design**);

by contrast, creating the conceptual schema (**logical** or **conceptual database design**) is the responsibility of the *data* administrator. And the DBMS is responsible, among other things, for implementing DDL and DML requests from the user. The DBMS is also responsible for providing some kind of **data dictionary** function.

Database systems can also be conveniently thought of as consisting of a **server** (the DBMS itself) and a set of **clients** (the applications). Client and server can and often will run on separate machines, thus providing one simple kind of **distributed processing**. In general, each server can serve many clients, and each client can access many servers. If the system provides total “transparency”—meaning that each client can behave as if it were dealing with a single server on a single machine, regardless of the overall physical state of affairs—then we have a true **distributed database system**.

EXERCISES

- 2.1 Draw a diagram of the database system architecture presented in this chapter (the ANSI/SPARC architecture).
- 2.2 Define the following terms:

backend	frontend
client	host language
conceptual DDL, schema, view	load
conceptual/internal mapping	logical database design
data definition language	internal DDL, schema, view
data dictionary	physical database design
data manipulation language	planned request
data sublanguage	reorganization
DB/DC system	server
DC manager	storage structure definition
distributed database	unload/reload
distributed processing	unplanned request
external DDL, schema, view	user interface
external/conceptual mapping	utility
- 2.3 Explain the sequence of steps involved in retrieving a particular external record occurrence.
- 2.4 List the major functions performed by the DBMS.
- 2.5 Distinguish between logical and physical data independence.
- 2.6 What do you understand by the term *metadata*?
- 2.7 List the major functions performed by the DBA.
- 2.8 Distinguish between the DBMS and a file management system.
- 2.9 Give some examples of vendor-provided tools.

2.10 Give some examples of database utilities.

2.11 Examine any database system that might be available to you. Try to map that system to the ANSI/SPARC architecture as described in this chapter. Does it cleanly support the three levels of the architecture? How are the mappings between levels defined? What do the various DDLs (external, conceptual, internal) look like? What data sublanguage(s) does the system support? What host languages? Who performs the DBA function? Are there any security or integrity facilities? Is there a dictionary? Is it self-describing? What vendor-provided applications does the system support? What utilities? Is there a separate DC manager? Are there any distributed processing capabilities?

REFERENCES AND BIBLIOGRAPHY

The following references are all fairly old by now (with the exception of the last one), but they are still relevant to the concepts introduced in the present chapter.

2.1 ANSI/X3/SPARC Study Group on Data Base Management Systems: Interim Report, *FDT* (bulletin of ACM SIGMOD) 7, No. 2 (1975).

2.2 Dionysios C. Tschritzis and Anthony Klug (eds.): "The ANSI/X3/SPARC DBMS Framework: Report of the Study Group on Data Base Management Systems," *Information Systems 3* (1978).

References [2.1–2.2] are the Interim and Final Report, respectively, of the so-called ANSI/SPARC Study Group. The ANSI/X3/SPARC Study Group on Data Base Management Systems (to give it its full title) was established in late 1972 by the Standards Planning and Requirements Committee (SPARC) of the American National Standards Institute (ANSI) Committee on Computers and Information Processing (X3). (Some 25 years later, the name X3 was changed to NCITS—National Committee on Information Technology Standards.) The objectives of the Study Group were to determine which areas, if any, of database technology were appropriate for standardization, and to produce a set of recommendations for action in each such area. In working to meet these objectives, the Study Group took the position that *interfaces* were the only aspect of a database system that could possibly be suitable for standardization, and accordingly defined a generalized database system architecture, or framework, that emphasized the role of such interfaces. The Final Report provides a detailed description of that architecture and of some of the 42 (!) identified interfaces. The Interim Report is an earlier working document that is still of some interest; in some areas it provides additional detail.

2.3 J. J. van Griethuysen (ed.): "Concepts and Terminology for the Conceptual Schema and the Information Base," International Organization for Standardization (ISO) Technical Report ISO/TR 9007:1987(E) (March 1982; revised July 1987).

This document is the report of an ISO Working Group whose objectives included "the definition of concepts for conceptual schema languages." It includes an introduction to three competing candidates (more accurately, three *sets* of candidates) for an appropriate set of formalisms, and applies each of the three to a common example involving the activities of a hypothetical car registration authority. The three sets of contenders are (1) "entity-attribute-relationship" schemes, (2) "binary relationship" schemes, and (3) "interpreted predicate logic" schemes. The report also includes a discussion of the fundamental concepts underlying the notion of the conceptual schema, and suggests some principles as a basis for implementation of a system that properly

supports that notion. Heavy going in places, but an important document for anyone seriously interested in the conceptual level of the system.

- 2.4** William Kent: *Data and Reality*. Amsterdam, Netherlands: North-Holland / New York, N.Y.: Elsevier Science (1978).

A stimulating and thought-provoking discussion of the nature of information, and in particular of the conceptual schema. "This book projects a philosophy that life and reality are at bottom amorphous, disordered, contradictory, inconsistent, nonrational, and nonobjective" (excerpt from the final chapter). The book can be regarded in large part as a compendium of real-world problems that (it is suggested) existing database formalisms—in particular, formalisms that are based on conventional record-like structures, which includes the relational model—have difficulty in dealing with. Recommended.

- 2.5** Odysseas G. Tsatalos, Marvin H. Solomon, and Yannis E. Ioannidis: "The GMAP: A Versatile Tool for Physical Data Independence," Proc. 20th Int. Conf. on Very Large Data Bases, Santiago, Chile (September 1994).

GMAP stands for *Generalized Multi-Level Access Path*. The authors of the paper note correctly that today's database products "force users to frame their queries in terms of a logical schema that is directly tied to physical structures," and hence are rather weak on physical data independence. In their paper, therefore, they propose a conceptual/internal mapping language (to use the terminology of the present chapter) that can be used to specify far more kinds of mappings than are typically supported in products today. Given a particular "logical schema," the language (which is based on relational algebra—see Chapter 6—and is therefore declarative, not procedural, in nature) allows the specification of numerous different physical schemas, all of them formally derived from that logical schema. Among other things, those physical schemas can include vertical and horizontal partitioning (see Chapter 20), any number of physical access paths, clustering, and controlled redundancy.

The paper also gives an algorithm for transforming user operations against the logical schema into equivalent operations against the physical schema. A prototype implementation shows that the DBA can tune the physical schema to "achieve significantly better performance than is possible with conventional techniques."

An Introduction to Relational Databases

3.1 INTRODUCTION

As explained in Chapter 1, the emphasis in this book is very much on relational systems. In particular, Part II covers the theoretical foundations of such systems (the relational model) in considerable depth. The purpose of the present chapter is just to give a preliminary, intuitive, and very informal introduction to the material to be addressed in Part II (and to some extent in subsequent parts too), in order to pave the way for a better understanding of those later parts of the book. Most of the topics mentioned will be discussed more formally, and in much more detail, in those later chapters.

3.2 AN INFORMAL LOOK AT THE RELATIONAL MODEL

We have already said several times that relational systems are based on a formal foundation, or theory, called *the relational model of data*. Intuitively, what this statement means (among other things) is that in such a system:

1. **Structural aspect:** The data in the database is perceived by the user as tables, and nothing but tables;
2. **Integrity aspect:** Those tables satisfy certain integrity constraints (to be discussed toward the end of this section);
3. **Manipulative aspect:** The operators available to the user for manipulating those tables—e.g., for purposes of data retrieval—are operators that derive tables from tables. Of those operators, three particularly important ones are *restrict*, *project*, and *join*.

A simple relational database, the departments and employees database, is shown in Fig. 3.1. As you can see, that database is indeed “perceived as tables” (and the meanings of those tables are intended to be self-evident).

DEPT	DEPT#	DNAME	BUDGET
	D1	Marketing	10M
	D2	Development	12M
	D3	Research	5M

EMP	EMP#	ENAME	DEPT#	SALARY
	E1	Lopez	D1	40K
	E2	Cheng	D1	42K
	E3	Finzi	D2	30K
	E4	Saito	D2	35K

Fig. 3.1 The departments and employees database (sample values)

Fig. 3.2 shows some sample restrict, project, and join operations against the database of Fig. 3.1. Here are (very loose!) definitions of those operations:

- The **restrict** operation (also known as select) extracts specified rows from a table.
- The **project** operation extracts specified columns from a table.
- The **join** operation joins two tables together on the basis of common values in a common column.

Restrict:	Result:	DEPT#	DNAME	BUDGET
DEPTs where BUDGET > 8M		D1	Marketing	10M
		D2	Development	12M

Project:	Result:	DEPT#	BUDGET
DEPTs over DEPT#, BUDGET		D1	10M
		D2	12M
		D3	5M

Join:	DEPTs and EMPs over DEPT#					
Result:	DEPT#	DNAME	BUDGET	EMP#	ENAME	SALARY
	D1	Marketing	10M	E1	Lopez	40K
	D1	Marketing	10M	E2	Cheng	42K
	D2	Development	12M	E3	Finzi	30K
	D2	Development	12M	E4	Saito	35K

Fig. 3.2 Restrict, project, and join (examples)

Of the three examples, the only one that seems to need any further explanation is the last one, the join example. Observe first that the two tables DEPT and EMP do indeed have a common column, DEPT#, so they can be joined together on the basis of common values in that column. A given row from table DEPT will join to a given row in table EMP (to yield a row of the result table) if and only if the two rows in question have a common DEPT# value. For example, the DEPT and EMP rows

DEPT#	DNAME	BUDGET	EMP#	ENAME	DEPT#	SALARY
D1	Marketing	10M	E1	Lopez	D1	40K

(column names shown for explicitness) join together to produce the result row

DEPT#	DNAME	BUDGET	EMP#	ENAME	SALARY
D1	Marketing	10M	E1	Lopez	40K

because they have the same value, D1, in the common column. Note that the common value appears once, not twice, in the result row. The overall result of the join contains all possible rows that can be obtained in this manner (and no other rows). Note in particular that since no EMP row has a DEPT# value of D3 (i.e., no employee is currently assigned to that department), no row for D3 appears in the result, even though there is a row for D3 in table DEPT.

Now, one point that Fig. 3.2 clearly shows is that *the result of each of the three operations is another table* (in other words, the operators are indeed “operators that derive tables from tables,” as previously stated). This is the **closure** property of relational systems, and it is very important. Basically, because the output of any operation is the same kind of object as the input—they are all tables—so *the output from one operation can become input to another*. Thus it is possible, e.g., to take a projection of a join, or a join of two restrictions, or a restriction of a projection, etc., etc. In other words, it is possible to write *nested expressions*—i.e., expressions in which the operands themselves are represented by general expressions, instead of just by simple table names. This fact in turn has numerous important consequences, as we will see later (both in this chapter and in many subsequent ones).

By the way, when we say that the output from each operation is another table, it is important to understand that we are talking *from a conceptual point of view*. We do not mean to imply that the system actually has to materialize the result of every individual operation in its entirety. For example, suppose we are trying to compute a restriction of a join. Then, as soon as a given row of the join is formed, the system can immediately test that row against the specified restriction condition to see whether it belongs in the final result, and immediately discard it if not. In other words, the intermediate result that is the output from the join might never exist as a fully materialized table in its own right at all. As a general rule, in fact, the system tries very hard *not* to materialize intermediate results in their entirety, for obvious performance reasons. *Note:* If intermediate results are fully materialized,

the overall expression evaluation strategy is called (unsurprisingly) *materialized evaluation*; if intermediate results are passed piecemeal to subsequent operations, it is called *pipelined evaluation*.

Another point that Fig. 3.2 also clearly illustrates is that the operations are all **set-at-a-time**, not row-at-a-time; that is, the operands and results are whole tables, not just single rows, and tables contain *sets* of rows. (A table containing a set of just one row is legal, of course; so too is an *empty* table, i.e., one containing no rows at all.) For example, the join in Fig. 3.2 operates on two tables of three and four rows respectively, and returns a result table of four rows. By contrast, the operations in nonrelational systems are typically at the row- or record-at-a-time level; thus, this *set-processing capability* is a major distinguishing characteristic of relational systems (see further discussion in Section 3.5 below).

Let us return to Fig. 3.1 for a moment. There are a couple of additional points to be made in connection with the sample database of that figure:

- First, note that relational systems require only that the database be *perceived by the user* as tables. Tables are the **logical** structure in a relational system, not the physical structure. At the physical level, in fact, the system is free to store the data any way it likes—using sequential files, indexing, hashing, pointer chains, compression, etc.—provided only that it can map that stored representation to tables at the logical level. Another way of saying the same thing is that tables represent an *abstraction* of the way the data is physically stored—an abstraction in which numerous storage-level details, such as stored record placement, stored record sequence, stored data value representations, stored record prefixes, stored access structures such as indexes, and so forth, are all *hidden from the user*.

Incidentally, the term “logical structure” in the foregoing paragraph is intended to encompass both the conceptual and external levels, in ANSI/SPARC terms. The point is that—as explained in Chapter 2—the conceptual and external levels in a relational system will both be relational, but the internal or physical level will not be. In fact, relational theory as such has nothing to say about the internal level at all; it is, to repeat, concerned with how the database looks to the *user*.^{*} The only requirement is that, to repeat, whatever physical structure is chosen must fully implement the logical structure.

- Second, relational databases abide by a very nice principle, called **The Information Principle**: *The entire information content of the database is represented in one and only one way, namely as explicit values in column positions in rows in tables.* This method of representation is the *only* method available (at the logical level, of course) in a relational system. In particular, **there are no pointers** connecting one table to another. In Fig. 3.1, for example, there is a connection between the D1 row of table DEPT and the E1 row of table EMP, because employee E1 works in department D1; but that

^{*}It is an unfortunate fact that most of today's SQL products do not support this aspect of the theory properly. To be more specific, they typically support only rather restrictive conceptual/internal mappings (typically, they map one logical table directly to one stored file). As a consequence, they do not provide nearly as much physical data independence as relational technology is theoretically capable of.

connection is represented, not by a pointer, but by the appearance of the value D1 in the DEPT# position of the EMP row for E1. In nonrelational systems, by contrast (e.g. IMS or IDMS), such information is typically represented—as mentioned in Chapter 1—by some kind of *pointer* that is explicitly visible to the user.

Note: When we say there are no pointers in a relational database, we do not mean there cannot be pointers at the *physical level*—on the contrary, there certainly can, and indeed there almost certainly will. But, to repeat, all such physical storage details are concealed from the user in a relational system.

So much for the structural and manipulative aspects of the relational model; now we turn to the integrity aspect. Consider the departments and employees database of Fig. 3.1 once again. In practice, that database might be required to satisfy any number of integrity constraints—for example, employee salaries might have to be in the range 25K to 95K (say), department budgets might have to be in the range 1M to 15M (say), and so on. Certain of those constraints are of such major pragmatic importance, however, that they enjoy some special nomenclature. To be specific:

1. Each row in table DEPT must include a unique DEPT# value; likewise, each row in table EMP must include a unique EMP# value. We say, loosely, that columns DEPT# in table DEPT and EMP# in table EMP are the **primary keys** for their respective tables. (Recall from Chapter 1 that we indicate primary keys in our figures by double underlining.)
2. Each DEPT# value in table EMP must exist as a DEPT# value in table DEPT, to reflect the fact that every employee must be assigned to an existing department. We say, loosely, that column DEPT# in table EMP is a **foreign key**, referencing the primary key of table DEPT.

We close this section with a definition of the relational model, for purposes of subsequent reference (despite the fact that the definition is quite abstract and will not be very comprehensible at this stage). Briefly, **the relational model** consists of the following five components:

1. An open-ended collection of **scalar types** (including in particular the type *boolean* or *truth value*);
2. A **relation type generator** and an intended interpretation for such generated relation types;
3. Facilities for defining **relation variables** of such generated relation types;
4. A **relational assignment** operation for assigning relation values to such relation variables;
5. An open-ended collection of generic **relational operators** for deriving relation values from other relation values.

As you can see, the relational model is very much more than just “tables plus restrict, project, and join,” though it is often characterized in such a manner informally.

Note: You might be surprised to see no explicit mention of integrity constraints in the foregoing definition. The fact is, however, such constraints represent just one application (albeit a very important one) of the relational operators; that is, such constraints are formulated—conceptually, at any rate—in terms of those operators, as we will see in Chapter 8.

3.3 RELATIONS AND RELVARS

If it is true that a relational database is basically just a database in which the data is perceived as tables—and of course it *is* true—then a good question to ask is: Why exactly do we call such a database relational? The answer is simple (in fact, we mentioned it in Chapter 1): “Relation” is just a mathematical term for a table—to be precise, a table of a certain specific kind (details to be discussed in Chapter 5). Thus, for example, we can say that the departments and employees database of Fig. 3.1 contains two *relations*.

Now, in informal contexts it is usual to treat the terms “relation” and “table” as if they were synonymous; informally, indeed, the term “table” is used much more often than the term “relation.” But it is worth taking a moment to understand why the term “relation” was introduced in the first place. Briefly, the explanation is as follows:

- As we have seen, relational systems are based on the relational model. The relational model in turn is an abstract theory of data that is based on certain aspects of mathematics (mainly set theory and predicate logic).
- The principles of the relational model were originally laid down in 1969–70 by E. F. Codd, at that time a researcher in IBM. It was late in 1968 that Codd, a mathematician by training, first realized that the discipline of mathematics could be used to inject some solid principles and rigor into a field—database management—that, prior to that time, was all too deficient in any such qualities. Codd’s ideas were first widely disseminated in a now classic paper, “A Relational Model of Data for Large Shared Data Banks” (see reference [5.1] in Chapter 5).
- Since that time, those ideas—by now almost universally accepted—have had a wide-ranging influence on just about every aspect of database technology, and indeed on other fields as well, such as the fields of artificial intelligence, natural-language processing, and hardware system design.

Now, the relational model as originally formulated by Codd very deliberately made use of certain terms, such as the term “relation” itself, that were not familiar in IT circles at that time (even though the concepts in some cases were). The trouble was, many of the more familiar terms were very *fuzzy*—they lacked the precision necessary to a formal theory of the kind that Codd was proposing. For example, consider the term “record.” At different times and in different contexts that single term can mean either a record *occurrence* or a record *type*; a *logical* record or a *physical* record; a *stored* record or a *virtual* record; and perhaps other things as well. The relational model therefore does not use the term “record” at all; instead, it uses the term “tuple” (rhymes with “couple”), which can be given a very precise

definition. We do not give that definition here; for present purposes, it is sufficient to say that the term “tuple” corresponds approximately to the notion of a row (just as the term “relation” corresponds approximately to the notion of a table). When we move on to study the more formal aspects of relational systems in Part II, we will make use of the formal terminology, but in this chapter we are not trying to be so formal (for the most part, at any rate), and we will mostly stick to terms such as “row” and “column” that are reasonably familiar. However, one formal term we *will* use a lot from this point forward is the term “relation” itself.

We return to the departments and employees database of Fig. 3.1 once again in order to make another important point. The fact is, DEPT and EMP in that database are really *relation variables*—variables, that is, whose values are relation *values* (different relation values at different times). For example, suppose EMP currently has the value—the *relation* value, that is—shown in Fig. 3.1, and suppose we delete the row for Saito (employee number E4):

```
DELETE EMP WHERE EMP# = 'E4' ;
```

The result is shown in Fig. 3.3.

EMP	EMP#	ENAME	DEPT#	SALARY
	E1	Lopez	D1	40K
	E2	Cheng	D1	42K
	E3	Finzi	D2	30K

Fig. 3.3 Relation variable EMP after deleting E4 row

Conceptually, what has happened here is that the *old relation value of EMP* has been *replaced en bloc by an entirely new relation value*. Of course, the old value (with four rows) and the new one (with three) are very similar, but conceptually they *are* different values. Indeed, the delete operation in question is basically just shorthand for a certain **relational assignment** operation that might look something like this:

```
EMP := EMP MINUS ( EMP WHERE EMP# = 'E4' ) ;
```

(*Note:* Both the original DELETE and the equivalent assignment statement are expressed in a language called **Tutorial D** [3.3]. MINUS is the **Tutorial D** keyword for the relational *difference* operation—see Chapter 6.) As in all assignments, what is happening here, conceptually speaking, is that (a) the expression on the right-hand side is evaluated, and then (b) the result of that evaluation is assigned to the variable on the left-hand side (by definition, of course, that left-hand side must identify a *variable* specifically). As already stated, the net effect is thus to replace the “old” EMP value by a “new” one.

In analogous fashion, of course, relational INSERT and UPDATE operations are also basically shorthand for certain relational assignments.

Now, it is an unfortunate fact that much of the literature uses the term *relation* when what it really means is a *relation variable* (as well as when it means a *relation per se*—i.e., a *relation value*). Historically, however, this practice has certainly led to some confusion. Throughout this book, therefore, we will distinguish very carefully between relation variables and relations *per se*; following reference [3.3], in fact, we will use the term **relvar** as a convenient shorthand for *relation variable*, and we will take care to phrase our remarks in terms of relvars, not relations, when it really is relvars that we mean.

Note: We should warn you that the term *relvar* is not in common usage—but it should be! We really do feel it is important to be clear about the distinction between relations *per se* (i.e., relation values) and relation variables, even though earlier editions of this book failed to do so, and most of the current database literature still fails in this respect.

3.4 WHAT RELATIONS MEAN

In Chapter 1, we mentioned the fact that columns in relations have associated **data types**.^{*} And at the end of Section 3.2, we said that the relational model includes “an open-ended set of [data] types.” What this means (among other things) is that **users will be able to define their own types** (as well as being able to use system-defined or *builtin* types, of course). For example, we might define types as follows (**Tutorial D** syntax again; the ellipses “...” stand for portions of the definitions that are not germane to the present discussion):

```
TYPE EMP# ... ;
TYPE NAME ... ;
TYPE DEPT# ... ;
TYPE MONEY ... ;
```

Type EMP#, for example, can be regarded (among other things) as *the set of all possible employee numbers*; type NAME as *the set of all possible names*; and so on.

Now consider Fig. 3.4, which is basically the EMP portion of Fig. 3.1 expanded to show the column data types. As the figure indicates, every relation—to be more precise, every relation *value*—has two parts, a set of column-name:type-name pairs (the **heading**), together with a set of rows that conform to that heading (the **body**). *Note:* In practice we often ignore the type-name components of the heading, as indeed we have done in all of our examples prior to this point, but you should understand that, conceptually, they are always there.

Now, there is a very important (though perhaps unusual) way of thinking about relations, and that is as follows:

^{*}The more usual relational term for data types is domains, as we will see in Chapter 5.

EMP											
EMP#	:	EMP#	ENAME	:	NAME	DEPT#	:	DEPT#	SALARY	:	MONEY
E1			Lopez			D1					40K
E2			Cheng			D1					42K
E3			Finzi			D2					30K
E4			Saito			D2					35K

Fig. 3.4 Sample EMP relation value, with column types shown

- First, given a relation r , the heading of r denotes a certain **predicate** or truth-valued function;
- Second, as mentioned briefly in Chapter 1, each row in the body of r denotes a certain **true proposition**, obtained from the predicate by substituting certain *argument* values of the appropriate type for the *placeholders* or *parameters* of that predicate ("instantiating the predicate").

In the case of Fig. 3.4, for example, the predicate looks something like this:

Employee EMP# is named ENAME, works in department DEPT#, and earns salary SALARY

(the parameters are EMP#, ENAME, DEPT#, and SALARY, corresponding of course to the four EMP columns). And the corresponding true propositions are:

Employee E1 is named Lopez, works in department D1, and earns salary 40K

(obtained by substituting the EMP# value E1, the NAME value Lopez, the DEPT# value D1, and the MONEY value 40K for the appropriate parameters);

Employee E2 is named Cheng, works in department D1, and earns salary 42K

(obtained by substituting the EMP# value E2, the NAME value Cheng, the DEPT# value D1, and the MONEY value 42K for the appropriate parameters); and so on. In a nutshell, therefore:

- **Types** are (sets of) things we can talk about;
- **Relations** are (sets of) things we say about the things we can talk about.

(There is a nice analogy here that might help you appreciate and remember these important points: *Types are to relations as nouns are to sentences*.) Thus, in the example, the things we can talk about are employee numbers, names, department numbers, and money values, and the things we say are true utterances of the form "The employee with the specified employee number has the specified name, works in the specified department, and earns the specified salary."

It follows from all of the foregoing that:

- First, types and relations are both *necessary* (without types, we have nothing to talk about; without relations, we cannot say anything).
- Second, types and relations are *sufficient*, as well as necessary—i.e., we do not need anything else, logically speaking.

Note: It also follows that *types and relations are not the same thing*. It is an unfortunate fact that certain commercial products—not relational ones, by definition!—are confused over this very point. We will touch on this confusion in Chapter 25 (Section 25.2).

By the way, it is important to understand that *every* relation has an associated predicate, including relations that are derived from others by means of relational operators such as join. For example, the DEPT relation of Fig. 3.1 and the three result relations of Fig. 3.2 have predicates as follows:

- *DEPT*: “Department DEPT# is named DNAME and has budget BUDGET.”
- *Restriction of DEPT where BUDGET > 8M*: “Department DEPT# is named DNAME and has budget BUDGET, which is greater than eight million dollars.”
- *Projection of DEPT over DEPT# and BUDGET*: “Department DEPT# has some name and has budget BUDGET.”
- *Join of DEPT and EMP over DEPT#*: “Department DEPT# is named DNAME and has budget BUDGET and employee EMP# is named ENAME, works in department DEPT#, and earns salary SALARY.” Note that this predicate has six parameters, not seven—the two references to DEPT# denote the same parameter.

3.5 OPTIMIZATION

As explained in Section 3.2, relational operations such as restrict, project, and join are all *set-level* operations. As a consequence, relational languages are often said to be *nonprocedural*, on the grounds that users specify *what*, not *how*—i.e., they say what they want, without specifying a procedure for getting it. The process of “navigating” around the stored data in order to satisfy the user’s request is performed automatically by the system, not manually by the user. For this reason, relational systems are sometimes said to perform **automatic navigation**. In nonrelational systems, by contrast, navigation is generally the responsibility of the user. A striking illustration of the benefits of automatic navigation is shown in Fig. 3.5, which contrasts a certain SQL INSERT statement with the “manual navigation” code the user might have to write to achieve an equivalent effect in a nonrelational system (actually a CODASYL network system; the example is taken from the chapter on network databases in reference [1.5]). *Note:* The database is the well-known suppliers and parts database. Refer to Section 3.9 for further explanation.

Despite the remarks of the previous paragraph, it has to be said that *nonprocedural* is not a very satisfactory term, common though it is, because procedurality and nonprocedurality are

```

INSERT INTO SP ( S#, P#, QTY )
VALUES ( 'S4', 'P3', 1000 ) ;

MOVE 'S4' TO S# IN S
FIND CALC S
ACCEPT S-SP-ADDR FROM S-SP CURRENCY
FIND LAST SP WITHIN S-SP
while SP found PERFORM
    ACCEPT S-SP-ADDR FROM S-SP CURRENCY
    FIND OWNER WITHIN P-SP
    GET P
    IF P# IN P < 'P3'
        leave loop
    END-IF
    FIND PRIOR SP WITHIN S-SP
END-PERFORM
MOVE 'P3' TO P# IN P
FIND CALC P
ACCEPT P-SP-ADDR FROM P-SP CURRENCY
FIND LAST SP WITHIN P-SP
while SP found PERFORM
    ACCEPT P-SP-ADDR FROM P-SP CURRENCY
    FIND OWNER WITHIN S-SP
    GET S
    IF S# IN S < 'S4'
        leave loop
    END-IF
    FIND PRIOR SP WITHIN P-SP
END-PERFORM
MOVE 1000 TO QTY IN SP
FIND DB-KEY IS S-SP-ADDR
FIND DB-KEY IS P-SP-ADDR
STORE SP
CONNECT SP TO S-SP
CONNECT SP TO P-SP

```

Fig. 3.5 Automatic vs. manual navigation

not absolutes. The best that can be said is that some language *A* is either more or less procedural than some other language *B*. Perhaps a better way of putting matters would be to say that languages such as SQL are at a *higher level of abstraction* than languages such as C++ and COBOL (or data sublanguages such as are typically found in nonrelational DBMSs, come to that, as Fig. 3.5 shows). Fundamentally, it is this raising of the level of abstraction that is responsible for the increased productivity that relational systems can provide.

Deciding just how to perform the automatic navigation referred to above is the responsibility of a very important DBMS component called the **optimizer** (we mentioned this component briefly in Chapter 2). In other words, for each relational request from the user, it is the job of the optimizer to choose an efficient way to implement that request. By way of an example, let us suppose the user issues the following request (**Tutorial D** once again):

```
RESULT := ( EMP WHERE EMP# = 'E4' ) { SALARY } ;
```

Explanation: The expression in parentheses ("EMP WHERE ...") denotes a restriction of the current value of relvar EMP to just the row where EMP# is E4. The column name in braces ("SALARY") then causes the result of that restriction to be projected over the SALARY column. Finally, the assignment operation (":=") causes the result of that projection to be assigned to relvar RESULT. After that assignment, RESULT contains a single-column, single-row relation that contains employee E4's salary. (Incidentally, note that we are implicitly making use of the relational *closure* property in this example—we have written a nested relational expression on the right-hand side, in which the input to the projection is the output from the restriction.)

Now, even in this very simple example, there are probably at least two ways of performing the necessary data access:

1. By doing a physical sequential scan of (the stored version of) relvar EMP until the required data is found;
2. If there is an index on (the stored version of) the EMP# column—which in practice there probably will be, because EMP# values are supposed to be unique, and many systems in fact *require* an index in order to enforce uniqueness—then by using that index and thus going directly to the required data.

The optimizer will choose which of these two strategies to adopt. More generally, given any particular request, the optimizer will make its choice of strategy for implementing that request on the basis of considerations such as the following:

- Which relvars are referenced in the request;
- How big those relvars currently are;
- What indexes exist;
- How selective those indexes are;
- How the data is physically clustered on the disk;
- What relational operations are involved;

and so on. To repeat, therefore: Users specify only what data they want, not how to get to that data; the access strategy for getting to the data is chosen by the optimizer ("automatic navigation"). Users and user programs are thus independent of such access strategies, which is of course essential if physical data independence is to be achieved.

We will have a lot more to say about the optimizer in Chapter 17.

3.6 THE CATALOG

As explained in Chapter 2, every DBMS must provide a **catalog** or **dictionary** function. The catalog is the place where—among other things—all of the various schemas (external, conceptual, internal) and all of the corresponding mappings (external/conceptual, conceptual/internal) are kept. In other words, the catalog contains detailed information (sometimes

called **descriptor information** or **metadata**) regarding the various objects that are of interest to the system itself. Examples of such objects are relvars, indexes, users, integrity constraints, security constraints, and so on and so forth. Descriptor information is essential if the system is to do its job properly. For example, the optimizer uses catalog information about indexes and other physical storage structures, as well as much other information, to help it decide how to implement user requests. Likewise, the security subsystem uses catalog information about users and security constraints (see Chapter 16) to grant or deny such requests in the first place.

Now, one of the nice features of relational systems is that, in such a system, *the catalog itself consists of relvars* (more precisely, *system relvars*, so called to distinguish them from ordinary user ones). As a result, users can interrogate the catalog in exactly the same way they interrogate their own data. For example, the catalog will typically include two system relvars called **TABLE** and **COLUMN**, the purpose of which is to describe the tables (i.e., relvars) in the database and the columns in those tables. (We say “typically” because the catalog is not the same in every system; the differences arise because the catalog for a particular system necessarily contains a good deal of information that is specific to that system.) For the departments and employees database of Fig. 3.1, the **TABLE** and **COLUMN** relvars might look—in outline—as shown in Fig. 3.6.*

Note: As mentioned in Chapter 2, the catalog should normally be self-describing—i.e., it should include entries describing the catalog relvars themselves. No such entries are shown in Fig. 3.6, however. See Exercise 3.3 at the end of the chapter.

TABLE	TABNAME	COLCOUNT	ROWCOUNT	
	DEPT	3	3	
	EMP	4	4	
				
COLUMN	TABNAME	COLNAME		
	DEPT	DEPT#		
	DEPT	DNAME		
	DEPT	BUDGET		
	EMP	EMP#		
	EMP	ENAME		
	EMP	DEPT#		
	EMP	SALARY		
				

Fig. 3.6 Catalog for the departments and employees database (in outline)

*Note that the presence of a **ROWCOUNT** column in the figure suggests that **INSERT** and **DELETE** operations on the database will cause an update to the catalog as a side effect. In practice, **ROWCOUNT** might be updated only on request (e.g., when some utility is run), meaning the values might not always be current.

Now suppose some user of the departments and employees database wants to know exactly what columns relvar DEPT contains (obviously we are assuming that for some reason the user does not already have this information). Then the expression

```
( COLUMN WHERE TABNAME = 'DEPT' ) { COLNAME }
```

does the job. *Note:* If we had wanted to keep the result of this query in some more permanent fashion, we could have assigned the value of the expression to some other relvar, as in the example in the previous section. We will omit this final assignment step from most of our remaining examples (both here and in subsequent chapters).

Here is another example: “Which relvars include a column called EMP#?”

```
( COLUMN WHERE COLNAME = 'EMP#' ) { TABNAME }
```

Exercise: What does the following do?

```
( ( TABLE JOIN COLUMN )  
  WHERE COLCOUNT < 5 ) { TABNAME, COLNAME }
```

3.7 BASE RELVARS AND VIEWS

We have seen that, starting with a set of relvars such as DEPT and EMP, together with a set of relation values for those relvars, relational expressions allow us to obtain further relation values from those given ones—e.g., by joining two of the given ones together. It is time to introduce a little more terminology. The original (given) relvars are called **base relvars**, and their relation values are called **base relations**; a relation that is or can be obtained from those base relations by means of some relational expression is called a **derived** or **derivable** relation. *Note:* Base relvars are called *real* relvars in reference [3.3].

Now, relational systems obviously have to provide a means for creating the base relvars in the first place. In SQL, for example, this task is performed by the CREATE TABLE statement (“TABLE” here meaning, very specifically, a base relvar). And base relvars obviously have to be *named*—for example:

```
CREATE TABLE EMP ... ;
```

However, relational systems usually support another kind of named relvar also, called a **view**, whose value at any given time is a *derived* relation (and so a view can be thought of, loosely, as a **derived relvar**). The value of a given view at a given time is whatever results from evaluating a certain relational expression at that time; the relational expression in question is specified at the time the view in question is created. For example, the statement

```
CREATE VIEW TOEMP AS  
  ( EMP WHERE SALARY > 33K ) { EMP#, ENAME, SALARY } ;
```

might be used to define a view called TOPEMP. *Note:* For reasons that are unimportant at this juncture, this example is expressed in a mixture of SQL and **Tutorial D**.

When this statement is executed, the relational expression following the AS—the **view-defining expression**—is not evaluated but is merely “remembered” by the system in some way (actually by saving it in the catalog, under the specified name TOPEMP). To the user, however, it is now as if there really were a relvar in the database called TOPEMP, with current value as indicated in the unshaded portions (only) of Fig. 3.7 below. And the user should be able to operate on that view just as if it were a base relvar. *Note:* If (as suggested above) DEPT and EMP are thought of as *real* relvars, then TOPEMP might be thought of as a *virtual* relvar—i.e., a relvar that apparently exists in its own right, but in fact does not (its value at any given time depends on the value(s) of certain other relvar(s)).

TOPEMP	EMP#	ENAME	DEPT#	SALARY
	E1	Lopez	D1	40K
	E2	Cheng	D1	42K
	E3	Ferris	D2	30K
	E4	Saito	D2	35K

Fig. 3.7 TOPEMP as a view of EMP (unshaded portions)

Note carefully, however, that although we say that the value of TOPEMP is the relation that would result if the view-defining expression were evaluated, we do not mean to suggest that we now have a *separate copy* of the data; i.e., we do not mean to suggest that the view-defining expression actually *is* evaluated. On the contrary, the view is effectively just a *window* into the underlying base relvar EMP. As a consequence, any changes to that underlying relvar will be automatically and instantaneously visible through that window (assuming they lie within the unshaded portion, of course). Likewise, changes to TOPEMP will automatically and instantaneously be applied to relvar EMP, and hence be visible through the window (see later for an example).

Here is a sample retrieval operation against view TOPEMP:

`( TOPEMP WHERE SALARY < 42K ) { EMP#, SALARY }`

Given the sample data of Fig. 3.7, the result will look like this:

EMP#	SALARY
E1	40K
E4	35K

Conceptually, operations against a view like the retrieval operation just shown are handled by replacing references to the view name by the view-defining expression (i.e.,

the expression that was saved in the catalog). In the example, therefore, the original expression

```
( TOPEMP WHERE SALARY < 42K ) { EMP#, SALARY }
```

is modified by the system to become

```
( ( EMP WHERE SALARY > 33K ) { EMP#, ENAME, SALARY } )  
  WHERE SALARY < 42K ) { EMP#, SALARY }
```

(we have italicized the view name in the original expression and the replacement text in the modified version). The modified expression can then be simplified to just

```
( EMP WHERE SALARY > 33K AND SALARY < 42K ) { EMP#, SALARY }
```

(see Chapter 17), which when evaluated yields the result shown earlier. In other words, the original operation against the view is effectively converted into an equivalent operation against the underlying base relvar, and that equivalent operation is then executed in the normal way (more accurately, *optimized and* executed in the normal way).

By way of another example, consider the following DELETE operation:

```
DELETE TOPEMP WHERE SALARY < 42K ;
```

The DELETE that is actually executed looks something like this:

```
DELETE EMP WHERE SALARY > 33K AND SALARY < 42K ;
```

Now, the view TOPEMP is very simple, consisting as it does just of a row-and-column subset of a single underlying base relvar (loosely speaking). In principle, however, a view definition, since it is essentially just a named relational expression, can be *of arbitrary complexity* (it can even refer to other views). For example, here is a view whose definition involves a join of two underlying base relvars:

```
CREATE VIEW JOINEX AS  
( ( EMP JOIN DEPT ) WHERE BUDGET > 7M ) { EMP#, DEPT# } ;
```

We will return to the whole question of view definition and view processing in Chapter 9.

Incidentally, we can now explain the remark in Chapter 2, near the end of Section 2.2, to the effect that the term "view" has a rather specific meaning in relational contexts that is not identical to the meaning ascribed to it in the ANSI/SPARC architecture. At the external level of that architecture, the database is perceived as an "external view," defined by an external schema (and different users can have different external views). In relational systems, by contrast, a view (as explained above) is, specifically, a *named, derived, virtual relvar*. Thus, the relational analog of an ANSI/SPARC "external view" is (typically) a collection of several relvars, each of which is a view in the relational sense, and the "external schema" consists of definitions of those views. (It follows that views in the relational sense are the relational model's way of providing **logical data independence**, though once

again it has to be said that today's commercial products are seriously deficient in this regard. See Chapter 9.)

Now, the ANSI/SPARC architecture is quite general and allows for arbitrary variability between the external and conceptual levels. In principle, even the *types* of data structures supported at the two levels could be different—for example, the conceptual level could be relational, while a given user could have an external view that was hierarchic. In practice, however, most systems use the same type of structure as the basis for both levels, and relational products are no exception to this rule—views are still relvars, just like the base relvars are. And since the same type of object is supported at both levels, the same data sublanguage (usually SQL) applies at both levels. Indeed, the fact that a view is a relvar is precisely one of the strengths of relational systems; it is important in just the same way that the fact that a subset is a set is important in mathematics. *Note:* SQL products and the SQL standard (see Chapter 4) often seem to miss this point, however, inasmuch as they refer repeatedly to “tables and views” (with the implication that a view is not a table). You are strongly advised *not* to fall into this common trap of taking “tables” (or “relvars”) to mean, specifically, *base* tables (or relvars) only.

There is one final point that needs to be made on the subject of base relvars vs. views, as follows. The base relvar vs. view distinction is frequently characterized thus:

- Base relvars “really exist,” in the sense that they represent data that is actually stored in the database;
- Views, by contrast, do not “really exist” but merely provide different ways of looking at “the real data.”

However, this characterization, though perhaps useful in an informal sense, does not accurately reflect the true state of affairs. It is true that users can *think* of base relvars as if they were physically stored; in a way, in fact, the whole point of relational systems is to allow users to think of base relvars as physically existing, while not having to concern themselves with how those relvars are actually represented in storage. But—and it is a big but!—this way of thinking should not be construed as meaning that a base relvar is physically stored in any kind of direct way (e.g., as a single stored file).^{*} As explained in Section 3.2, base relvars are best thought of as an *abstraction* of some collection of stored data—an abstraction in which all storage-level details are concealed. In principle, there can be an arbitrary degree of differentiation between a base relvar and its stored counterpart.

A simple example might help to clarify this point. Consider the departments and employees database once again. Most of today's relational systems would probably implement that database with two stored files, one for each of the two base relvars. But there is ab-

^{*}The following quote from a recent book displays several of the confusions being discussed here, as well as others discussed in Section 3.3 earlier: “[It] is important to make a distinction between stored relations, which are *tables*, and virtual relations, which are *views* . . . [We] shall use *relation* only where a table or a view could be used. When we want to emphasize that a relation is stored, rather than a view, we shall sometimes use the term *base relation* or *base table*.” The quote is, regrettably, not at all atypical.

solutely no logical reason why there should not be just one stored file of *hierarchic* stored records, each consisting of (a) the department number, name, and budget for some given department, together with (b) the employee number, name, and salary for each employee who happens to be in that department. In other words, the data can be physically stored in whatever way seems appropriate, but it always looks the same at the logical level.

3.8 TRANSACTIONS

Note: The topic of this section is not peculiar to relational systems. We cover it here nevertheless, because an understanding of the basic idea is needed in order to appreciate certain aspects of the material to come in Part II. However, our coverage at this point is deliberately not very deep.

In Chapter 1 we said that a transaction is a *logical unit of work*, typically involving several database operations. We also indicated that the user needs to be able to inform the system when distinct operations are part of the same transaction. The BEGIN TRANSACTION, COMMIT, and ROLLBACK operations are provided for this purpose. Basically, a transaction begins when a BEGIN TRANSACTION operation is executed, and terminates when a corresponding COMMIT or ROLLBACK operation is executed. For example (pseudocode):

```
BEGIN TRANSACTION ; /* move $$$ from account A to account B */
UPDATE account A ;      /* withdrawal */
UPDATE account B ;      /* deposit */
IF everything worked fine
    THEN COMMIT ;        /* normal end */
    ELSE ROLLBACK ;      /* abnormal end */
END IF ;
```

Points arising:

1. Transactions are guaranteed to be **atomic**—i.e., they are guaranteed (from a logical point of view) either to execute in their entirety or not to execute at all, even if (say) the system fails halfway through the process.
2. Transactions are also guaranteed to be **durable**, in the sense that once a transaction successfully executes COMMIT, its updates are guaranteed to be applied to the database, even if the system subsequently fails at any point. *Note:* Fundamentally, it is this durability property of transactions that makes the data in the database *persistent*, in the sense of Chapter 1.
3. Transactions are also guaranteed to be **isolated** from one another, in the sense that database updates made by a given transaction *T1* are not made visible to any distinct transaction *T2* until and unless *T1* successfully executes COMMIT. COMMIT causes updates made by the transaction to become visible to other transactions; such updates are said to be *committed*, and are guaranteed never to be canceled. If the transaction executes ROLLBACK instead, all updates made by the transaction are canceled (*rolled back*). In this latter case, the effect is as if the transaction never ran in the first place.

4. The interleaved execution of a set of concurrent transactions is (usually) guaranteed to be **serializable**, in the sense that it produces the same result as executing those same transactions one at a time in some unspecified serial order.

Refer to Chapters 14–15 for an extended discussion of all of the foregoing points and much else besides.

3.9 THE SUPPLIERS AND PARTS DATABASE

Our running example throughout most of this book is the well-known **suppliers and parts** database (maintained, as we saw in Chapter 1, by a fictitious company called KnowWare Inc.). The purpose of this section is to explain that database, in order to serve as a point of reference for later chapters. Fig. 3.8 shows a set of sample data values; subsequent examples will actually assume these specific values, where it makes any difference. Fig. 3.9 shows the database definition, expressed in (a slight variation on) **Tutorial D** once again. Note the primary and foreign key specifications in particular. Note too that (a) several columns have data types of the same name as the column in question; (b) the STATUS column and the two CITY columns are defined in terms of builtin types—INTEGER (integers) and CHAR (character strings of arbitrary length)—instead of user-defined ones. Note finally that there is an important point to be made regarding the column values as shown in Fig. 3.8, but we are not yet in a position to make that point; we will come back to it in Chapter 5, Section 5.2, in the subsection entitled “Possible Representations.”

S	S#	SNAME	STATUS	CITY	SP	S#	P#	QTY
	S1	Smith	20	London		S1	P1	300
	S2	Jones	10	Paris		S1	P2	200
	S3	Blake	30	Paris		S1	P3	400
	S4	Clark	20	London		S1	P4	200
	S5	Adams	30	Athens		S1	P5	100
P	P#	PNAME	COLOR	WEIGHT		S1	P6	100
	P1	Nut	Red	12.0		S2	P1	300
	P2	Bolt	Green	17.0		S2	P2	400
	P3	Screw	Blue	17.0		S3	P2	200
	P4	Screw	Red	14.0		S4	P2	200
	P5	Cam	Blue	12.0		S4	P4	300
	P6	Cog	Red	19.0		S4	P5	400

Fig. 3.8 The suppliers and parts database (sample values)

```

TYPE S# ... ;
TYPE NAME ... ;
TYPE P# ... ;
TYPE COLOR ... ;
TYPE WEIGHT ... ;
TYPE QTY ... ;

VAR S BASE RELATION
{ S#      S#,
  SNAME   NAME,
  STATUS  INTEGER,
  CITY    CHAR }
PRIMARY KEY { S# } ;

VAR P BASE RELATION
{ P#      P#,
  PNAME   NAME,
  COLOR   COLOR,
  WEIGHT  WEIGHT,
  CITY    CHAR }
PRIMARY KEY { P# } ;

VAR SP BASE RELATION
{ S#      S#,
  P#      P#,
  QTY     QTY }
PRIMARY KEY { S#, P# }
FOREIGN KEY { S# } REFERENCES S
FOREIGN KEY { P# } REFERENCES P ;

```

Fig. 3.9 The suppliers and parts database (data definition)

The database is meant to be understood as follows:

- Relvar S represents *suppliers*. Each supplier has a supplier number (S#), unique to that supplier; a supplier name (SNAME), not necessarily unique (though the SNAME values do happen to be unique in Fig. 3.8); a rating or status value (STATUS); and a location (CITY). We assume that each supplier is located in exactly one city.
- Relvar P represents *parts* (more accurately, *kinds* of parts). Each kind of part has a part number (P#), which is unique; a part name (PNAME); a color (COLOR); a weight (WEIGHT); and a location where parts of that type are stored (CITY). We assume—that it makes any difference—that part weights are given in pounds. We also assume that each kind of part comes in exactly one color and is stored in a warehouse in exactly one city.
- Relvar SP represents *shipments*. It serves in a sense to link the other two relvars together, logically speaking. For example, the first row in SP as shown in Fig. 3.8 links a specific supplier from S (namely, supplier S1) to a specific part from P (namely, part P1)—in other words, it represents a shipment of parts of kind P1 by the supplier called S1 (and the shipment quantity is 300). Thus, each shipment has a supplier number (S#), a part number (P#), and a quantity (QTY). We assume there can be at most one shipment at any given time for a given supplier and a given part; for a given shipment,

therefore, the combination of S# value and P# value is unique with respect to the set of shipments currently appearing in SP. *Note:* The sample values in Fig. 3.8 deliberately include one supplier, supplier S5, with no matching shipments at all.

We remark that (as already pointed out in Chapter 1, Section 1.3) suppliers and parts can be regarded as **entities**, and a shipment can be regarded as a **relationship** between a particular supplier and a particular part. As also pointed out in that section, however, a relationship is best regarded as just a special case of an entity. One advantage of relational databases is precisely that all entities, regardless of whether they are in fact relationships, are represented in the same uniform way—namely, by means of rows in relations, as the example indicates.

A couple of final remarks:

- First, the suppliers and parts database is of course extremely simple, much simpler than any real database is likely to be; most real databases will involve many more entities and relationships (and many more *kinds* of entities and relationships) than this one does. Nevertheless, it is at least adequate to illustrate most of the points that we need to make in the rest of the book, and (as already stated) we will use it as the basis for most—not all—of our examples in the next few chapters.
- Second, there is (of course) nothing wrong with using more descriptive names such as SUPPLIERS, PARTS, and SHIPMENTS in place of the rather terse names S, P, and SP used above; indeed, descriptive names are generally to be recommended in practice. But in the case of suppliers and parts specifically, the relvars are referenced so frequently in the chapters that follow that very short names seemed desirable. Long names tend to become irksome with much repetition.

3.10 SUMMARY

This brings us to the end of our short overview of relational technology. Obviously we have barely scratched the surface of what by now has become a very extensive subject, but the whole point of the chapter has been to serve as a gentle introduction to the much more comprehensive discussions to come. Even so, we have managed to cover quite a lot of ground. Here is a summary of the major topics we have discussed.

A **relational database** is a database that is perceived by its users as a collection of **relation variables**—i.e., *relvars*—or, more informally, **tables**. A **relational system** is a system that supports relational databases and operations on such databases, including in particular the operations **restrict**, **project**, and **join**. These operations, and others like them, are all **set-level**. The **closure** property of relational systems means that the output from every operation is the same kind of object as the input (they are all relations), which means we can write **nested relational expressions**. Relvars can be updated by means of the **relational assignment** operation; the familiar update operations **INSERT**, **UPDATE**, and **DELETE** can be regarded as shorthands for certain common relational assignments.

The formal theory underlying relational systems is called the **relational model of data**. The relational model is concerned with logical matters only, not physical matters. It addresses three principal aspects of data—data **structure**, data **integrity**, and data **manipulation**. The *structural* aspect has to do with relations *per se*; the *integrity* aspect has to do with (among other things) **primary and foreign keys**; and the *manipulative* aspect has to do with the operators (restrict, project, join, etc.). The **Information Principle** states that the entire information content of a relational database is represented in one and only one way, namely as explicit values in column positions in rows in relations.

Every relation has a **heading** and a **body**; the heading is a set of column-name:type-name pairs, the body is a set of rows that conform to the heading. The heading of a given relation can be regarded as a **predicate**, and each row in the body denotes a certain **true proposition**, obtained by substituting certain **argument** values of the appropriate type for the *placeholders* or **parameters** of the predicate. In other words, *types* are (sets of) things we can talk about, and *relations* are (sets of) things we say about the things we can talk about. Together, types and relations are **necessary** and **sufficient** to represent any data we like (at the logical level, that is).

The **optimizer** is the system component that determines how to implement user requests (which are concerned with *what*, not *how*). Since relational systems therefore assume responsibility for navigating around the stored database to locate the desired data, they are sometimes described as **automatic navigation** systems. Optimization and automatic navigation are prerequisites for **physical data independence**.

The **catalog** is a set of system relvars that contain **descriptors** for the various items that are of interest to the system (base relvars, views, indexes, users, etc.). Users can interrogate the catalog in exactly the same way they interrogate their own data.

The original relvars in a given database are called **base relvars**, and their values are called **base relations**; a relation that is obtained from those base relations by means of some relational expression is called a **derived** relation (collectively, base and derived relations are sometimes referred to as **expressible** relations). A **view** is a relvar whose value at any given time is such a derived relation (loosely, it can be thought of as a **derived relvar**); the value of such a relvar at any given time is whatever results from evaluating the associated **view-defining expression**. Note, therefore, that base relvars have *independent existence*, but views do not—they depend on the applicable base relvars. (Another way of saying the same thing is that base relvars are **autonomous**, but views are not.) Users can operate on views in exactly the same way as they operate on base relvars (at least in theory). The system implements operations on views by replacing references to the name of the view by the view-defining expression, thereby converting the operation into an equivalent operation on the underlying base relvar(s).

A **transaction** is a *logical unit of work*, typically involving several database operations. A transaction begins when **BEGIN TRANSACTION** is executed and terminates when **COMMIT** (normal termination) or **ROLLBACK** (abnormal termination) is executed. Transactions are **atomic**, **durable**, and **isolated** from one another. The interleaved execution of a set of concurrent transactions is (usually) guaranteed to be **serializable**.

Finally, the base example for most of the book is **the suppliers and parts database**. It is worth taking the time to familiarize yourself with that example now, if you have not already done so; that is, you should at least know which relvars have which columns and what the primary and foreign keys are (it is not so important to know exactly what the sample data values are!).

EXERCISES

3.1 Define the following terms:

automatic navigation	primary key
base relvar	projection
catalog	proposition
closure	relational database
commit	relational DBMS
derived relvar	relational model
foreign key	restriction
join	rollback
optimization	set-level operation
predicate	view

3.2 Sketch the contents of the catalog relvars TABLE and COLUMN for the suppliers and parts database.

3.3 As explained in Section 3.6, the catalog is self-describing—i.e., it includes entries for the catalog relvars themselves. Extend Fig. 3.6 to include the necessary entries for the TABLE and COLUMN relvars themselves.

3.4 Here is a query on the suppliers and parts database. What does it do?

```
RESULT := ( ( S JOIN SP ) WHERE P# = 'P2' ) { S#, CITY } ;
```

Note: Actually there is a slight problem with this query, having to do with the data type of the P# column. We will come back to this point in Chapter 5, Section 5.2 (subsection “Type Conversions”). Analogous remarks apply to the next exercise also.

3.5 Suppose the expression on the right-hand side of the assignment in Exercise 3.4 is used in a view definition:

```
CREATE VIEW V AS
  ( ( S JOIN SP ) WHERE P# = 'P2' ) { S#, CITY } ;
```

Now consider the query

```
RESULT := ( V WHERE CITY = 'London' ) { S# } ;
```

What does this query do? Show what is involved on the part of the DBMS in processing this query.

3.6 What do you understand by the terms (transaction) atomicity, durability, isolation, and serializability?

REFERENCES AND BIBLIOGRAPHY

3.1 E. F. Codd: "Relational Database: A Practical Foundation for Productivity," *CACM* 25, No. 2 (February 1982). Republished in Robert L. Ashenhurst (ed.), *ACM Turing Award Lectures: The First Twenty Years 1966–1985*. Reading, Mass.: Addison-Wesley (*ACM Press Anthology Series*, 1987).

This is the paper Codd presented on the occasion of his receiving the 1981 ACM Turing Award for his work on the relational model. It discusses the well-known *application backlog* problem. To paraphrase: "The demand for computer applications is growing fast—so fast that information systems departments (whose responsibility it is to provide those applications) are lagging further and further behind in their ability to meet that demand." There are two complementary ways of attacking this problem:

1. Provide IT professionals with new tools to increase their productivity;
2. Allow end users to interact directly with the database, thus bypassing the IT professional entirely.

Both approaches are needed, and in this paper Codd gives evidence to suggest that the necessary foundation for both is provided by relational technology.

3.2 C. J. Date: "Why Relational?", in *Relational Database Writings 1985–1989*. Reading, Mass.: Addison-Wesley (1990).

An attempt to provide a succinct yet reasonably comprehensive summary of the major advantages of relational systems. The following observation from the paper is worth repeating here: Among all the numerous advantages of "going relational," there is one in particular that cannot be overemphasized, and that is *the existence of a sound theoretical base*. To quote:

"... relational really is different. It is different because it is not *ad hoc*. Older systems, by contrast, were *ad hoc*; they may have provided solutions to certain important problems of their day, but they did not rest on any solid theoretical base. Relational systems, by contrast, do rest on such a base . . . which means [they] are *rock solid*."

"Thanks to this solid foundation, relational systems behave in well-defined ways; and (possibly without realizing the fact) users have a simple model of that behavior in their mind, one that enables them to predict with confidence what the system will do in any given situation. There are (or should be) no surprises. This predictability means that user interfaces are easy to understand, document, teach, learn, use, and remember."

3.3 C. J. Date and Hugh Darwen: *Foundation for Object/Relational Databases: The Third Manifesto*. Reading, Mass.: Addison-Wesley (1998). See also the introductory overview paper "The Third Manifesto: Foundation for Object/Relational Databases," in C. J. Date, Hugh Darwen, and David McGoveran, *Relational Database Writings 1994–1997*. Reading, Mass.: Addison-Wesley (1998).

The Third Manifesto is a detailed, formal, and rigorous proposal for the future direction of databases and DBMSs. It can be seen as an abstract blueprint for the design of a DBMS and the language interface to such a DBMS. It is based on the classical core concepts **type**, **value**, **variable**,

and **operator**. For example, we might have a type INTEGER; the integer "3" might be a value of that type; N might be a variable of that type, whose value at any given time is some integer value (i.e., some value of that type); and "+" might be an operator that applies to integer values (i.e., to values of that type).

ANSWERS TO SELECTED EXERCISES

3.3 Fig. 3.10 below shows the entries for the TABLE and COLUMN relvars (only—the entries for the user's own relvars are omitted). It is obviously not possible to give precise COLCOUNT and ROWCOUNT values.

3.4 The query retrieves supplier number and city for suppliers who supply part P2.

3.5 The meaning of the query is "Get supplier numbers for London suppliers who supply part P2." The first step in processing the query is to replace the name V by the expression that defines V, giving:

```
{ ( { ( S JOIN SP ) WHERE P# = 'P2' ) { S#, CITY } }
  WHERE CITY = 'London' ) { S# }
```

This simplifies to:

```
{ ( S WHERE CITY = 'London' ) JOIN
  ( SP WHERE P# = 'P2' ) ) { S# }
```

For further discussion and explanation, see Chapters 9 and 17.

TABLE	TABNAME	COLCOUNT	ROWCOUNT	
	TABLE	(>3)	(>2)	
	COLUMN	(>2)	(>5)	
				
COLUMN	TABNAME	COLNAME		
	TABLE	TABNAME		
	TABLE	COLCOUNT		
	TABLE	ROWCOUNT		
	COLUMN	TABNAME		
	COLUMN	COLNAME		
				

Fig. 3.10 Catalog entries for TABLE and COLUMN themselves (in outline)

An Introduction to SQL

4.1 INTRODUCTION

As noted in Chapter 1, SQL is the standard language for dealing with relational databases, and it is supported by just about every product on the market. SQL was originally developed in IBM Research in the early 1970s (see references [4.8–4.9] and [4.28]; it was first implemented on a large scale in an IBM prototype called System R (see references [4.1–4.3] and [4.11–4.13]), and subsequently reimplemented in numerous commercial products from IBM (see reference [4.20]) and many other vendors. In this chapter we present an introduction to the SQL language; additional aspects, having to do with such matters as integrity, security, etc., are described in subsequent chapters devoted to those topics. All of our discussions are based (except where otherwise indicated) on the current version of the standard known informally as **SQL/92**, also as *SQL-92* or just *SQL2* [4.22–4.23]; the official name is **International Standard Database Language SQL (1992)**.

Note: We should immediately add that work is nearing completion on **SQL3**, the proposed follow-on to the current standard; ratification is expected in late 1999. By the time this book appears in print, therefore, the current standard might possibly be “SQL/99,” not SQL/92. However, we felt it would be too confusing to base our discussions on SQL3, since—fairly obviously—no product yet supports it; we therefore decided to discuss SQL3 separately in an appendix (Appendix B). In any case, we should also point out that no product today supports even SQL/92 in its entirety;* instead, products typically support what might be called “a superset of a subset” of SQL/92 (i.e., most of them fail to support certain aspects of the standard and yet go beyond the standard in certain other respects). For example, IBM’s DB2 product does not currently support all of the SQL/92 integrity features, but it does go beyond SQL/92 in its rules regarding the updating of views.

A few additional preliminary remarks:

- SQL was originally intended to be a “data sublanguage” specifically. With the incorporation of the *Persistent Stored Modules* feature (PSM) into the standard in late 1996, however, SQL became computationally complete (it now includes statements such as

*In fact no product possibly could support SQL/92 in its entirety, because SQL/92 as currently specified contains numerous gaps, mistakes, and inconsistencies. See reference [4.19] for a detailed discussion of this point.

CALL, RETURN, SET, CASE, IF, LOOP, LEAVE, WHILE, and REPEAT, as well as several related features such as variables and exception handlers). As a consequence, there is no longer any need to combine SQL with some distinct “host” language in order to develop a complete application. However, we choose not to discuss PSM in any detail in this book.

- You will not be surprised to learn that SQL uses the term **table** in place of both of the terms *relation* and *relvar* (see Chapter 3). For consistency with the SQL standard and SQL products, therefore, we will do likewise in this chapter (and elsewhere in this book whenever we are concerned with SQL specifically). Also, SQL does not use the terms *heading* and *body* (of a table or relation).
- SQL is an enormous language. The standard document [4.22] itself is well over 600 pages long (and the SQL3 specifications are over twice that size). As a consequence, it is not possible in a book of this nature to treat the subject exhaustively; all we can hope to do is describe major aspects in a reasonably comprehensive manner, but you are warned that our discussions are necessarily superficial. In particular, we have not hesitated to omit material that is irrelevant to the purpose at hand, nor to make simplifications in the interest of brevity. More complete (but still tutorial) descriptions can be found in references [4.4], [4.19], and [4.27].
- Finally, it has to be said that (as indicated at various points in Chapters 1–3) SQL is very far from being the perfect relational language—it suffers from sins of both omission and commission. Nevertheless, it is the standard, it is supported by just about every product on the market, and every database professional needs to know something about it. Hence the coverage in this book.

4.2 OVERVIEW

SQL includes both data definition operations and data manipulation operations. We consider **definitional** operations first. Fig. 4.1 gives an SQL definition for the suppliers and parts database (compare and contrast Fig. 3.9 in Chapter 3). As you can see, the definition includes one **CREATE TABLE** statement for each of the three base tables (as noted in Chapter 3, the keyword **TABLE** in **CREATE TABLE** means a base table specifically). Each such **CREATE TABLE** statement specifies the name of the base table to be created, the names and data types of the columns of that table, and the primary key and any foreign keys in that table (possibly some additional information also, not illustrated in Fig. 4.1). A couple of syntactic points arising:

- Note that we often make use of the “#” character in (e.g.) names of columns, but in fact that character is not legal in SQL/92.
- We use the semicolon “;” as a statement terminator. Whether SQL/92 actually uses such terminators depends on the context. The specifics are beyond the scope of this book.

```

CREATE TABLE S
(
  S#      CHAR(5),
  SNAME   CHAR(20),
  STATUS  NUMERIC(5),
  CITY    CHAR(15),
  PRIMARY KEY ( S# ) ) ;

CREATE TABLE P
(
  P#      CHAR(6),
  PNAME   CHAR(20),
  COLOR   CHAR(6),
  WEIGHT  NUMERIC(5,1),
  CITY    CHAR(15),
  PRIMARY KEY ( P# ) ) ;

CREATE TABLE SP
(
  S#      CHAR(5),
  P#      CHAR(6),
  QTY     NUMERIC(9),
  PRIMARY KEY ( S#, P# ),
  FOREIGN KEY ( S# ) REFERENCES S,
  FOREIGN KEY ( P# ) REFERENCES P ;

```

Fig. 4.1 The suppliers and parts database (SQL definition)

One important difference between Fig. 4.1 and its counterpart (Fig. 3.9) in Chapter 3 is that Fig. 4.1 includes nothing corresponding to the type definitions (i.e., TYPE statements) of Fig. 3.9. The reason is, of course, that SQL does not permit users to define their own types;* thus, columns must be defined in terms of *builtin* (system-defined) types only. SQL supports the following more or less self-explanatory builtin types:

CHARACTER [VARYING] (n)	INTEGER	DATE
BIT [VARYING] (n)	SMALLINT	TIME
NUMERIC (p,q)	FLOAT (p)	TIMESTAMP
DECIMAL (p,q)		INTERVAL

A number of defaults, abbreviations, and alternative spellings—e.g., CHAR for CHARACTER—are also supported; we omit the details here. Also, the square brackets “[” and “]” on CHARACTER and BIT are meant to signify that the material they enclose is optional (as is normal with BNF notation, of course). Observe finally that SQL requires certain *lengths* or *precisions* to be stated for certain types (e.g., CHAR), as our hypothetical syntax of Chapter 3 did not. In fact, SQL apparently regards those lengths and precisions as part of the type (implying that, e.g., CHAR(3) and CHAR(4) are different types); we believe it is better to regard them as *integrity constraints* (and so we do—see Chapter 8, especially Exercise 8.4).

Having defined the database, we can now start operating on it by means of the SQL **manipulative** operations SELECT, INSERT, UPDATE, and DELETE. In particular, we

*It does permit them to define what it calls *domains*, but those “domains” are not really domains—i.e., types—in the relational sense (see Chapter 5). *Note:* User-defined types are supported in SQL3, however (see Appendix B).

can perform relational restrict, project, and join operations on the data, in each case by using the SQL data manipulation statement **SELECT**. Fig. 4.2 shows how certain restrict, project, and join operations might be formulated in SQL. *Note:* The join example in that figure illustrates the point that **qualified names** (e.g., **S.S#**, **SP.S#**) are sometimes necessary in SQL to “disambiguate” column references. The broad rule is that qualified names are always acceptable, but unqualified names are also acceptable so long as they cause no ambiguity.

We remark that SQL also supports a shorthand form of the **SELECT** clause as illustrated by the following example:

```
SELECT * -- or "SELECT S.*" (i.e., the "*" can be qualified)
FROM S ;
```

The result is a copy of the entire **S** table; the star or asterisk is shorthand for a comma-separated list of all column names in the table(s) referenced in the **FROM** clause, in the left-to-right order in which those column(s) are defined within those table(s). Notice the **comment** in this example, incidentally (introduced with a double hyphen and terminated with a newline character). *Note:* The expression **SELECT * FROM T**, where **T** is a table name, can be further abbreviated to just **TABLE T**.

The **SELECT** statement is discussed at much greater length in Chapter 7 (Section 7.7).

Restrict:
Result:

S#	P#	QTY
S1	P5	100
S1	P6	100

SELECT S#, P#, QTY
FROM SP
WHERE QTY < 150 ;

Project:
Result:

S#	CITY
S1	London
S2	Paris
S3	Paris
S4	London
S5	Athens

SELECT S#, CITY
FROM S ;

Join:

SELECT S.S#, SNAME, STATUS, CITY, P#, QTY
FROM S, SP
WHERE S.S# = SP.S# ;

Result:

S#	SNAME	STATUS	CITY	P#	QTY
S1	Smith	20	London	P1	300
S1	Smith	20	London	P2	200
S1	Smith	20	London	P3	400
..	*****	..	*****	..	***
S4	Clark	20	London	P5	400

Fig. 4.2 Restrict, project, and join examples in SQL

Turning now to **update** operations: Examples of the SQL INSERT, UPDATE, and DELETE statements have already been given in Chapter 1, but those examples deliberately involved single-row operations only. Like SELECT, however, **INSERT**, **UPDATE**, and **DELETE** are all *set-level* operations, in general (and some of the exercises and answers in Chapter 1 did in fact illustrate this point). Here are some set-level update examples for the suppliers and parts database:

```
INSERT
INTO   TEMP ( P#, WEIGHT )
SELECT P#, WEIGHT
FROM   P
WHERE  COLOR = 'Red' ;
```

This example assumes that we have already created another table TEMP with two columns, P# and WEIGHT. The INSERT statement inserts into that table part numbers and corresponding weights for all red parts.

```
UPDATE S
SET    STATUS = STATUS * 2
WHERE  CITY = 'Paris' ;
```

This UPDATE statement doubles the status for all suppliers in Paris.

```
DELETE
FROM   SP
WHERE  P# = *
```

This DELETE statement deletes all shipments for part P2.

Note: SQL does not include a direct analog of the **relational assignment** operation. However, we can simulate that operation by first deleting all rows from the target table and then performing an INSERT ... SELECT ... (as in the first example above) into that table.

4.3 THE CATALOG

The SQL standard does include specifications for a standard catalog called the **Information Schema**. In fact, the conventional terms “catalog” and “schema” are both used in SQL, but with highly SQL-specific meanings; loosely speaking, an SQL **catalog** consists of the descriptors for an individual database,* and an SQL **schema** consists of the descriptors for that portion of that database that belongs to some individual user. In other words, there can be any number of catalogs (one per database), each divided up into any number of schemas. However, each catalog is required to include exactly one schema called INFORMATION _SCHEMA, and from the user’s perspective it is that schema (as already indicated) that performs the normal catalog function.

*In the interests of accuracy, we should mention that there is actually no such thing as a “database” in the SQL standard! Exactly what the collection of data is called that is described by a catalog is implementation-defined. However, it is not unreasonable to think of it as a database.

The Information Schema thus consists of a set of SQL tables whose contents effectively echo, in a precisely defined way, all of the definitions from all of the other schemas in the catalog in question. More precisely, the Information Schema is defined to contain a set of views of a hypothetical "Definition Schema." The implementation is not required to support the Definition Schema as such, but it is required (a) to support *some* kind of "Definition Schema," and (b) to support views of that "Definition Schema" that do look like those of the Information Schema. Points arising:

1. The rationale for stating the requirement in terms of two separate pieces (a) and (b) as just described is as follows. First, existing products certainly do support something akin to the "Definition Schema." However, those "Definition Schemas" vary widely from one product to another (even when the products in question come from the same vendor). Hence the idea of requiring only that the implementation support certain predefined views of its "Definition Schema" does make sense.
2. We should really say "an" (not "the") Information Schema, since as we have seen there is one such schema in every catalog. In general, therefore, the totality of data available to a given user will *not* be described by a single Information Schema. For simplicity, however, we will continue to talk as if there were just one.

It is not worth going into great detail on the content of the Information Schema here; instead, we simply list some of the more important Information Schema views, in the hope that their names alone will be sufficient to give some idea of what those views contain. We should say, however, that the TABLES view contains information for *all* named tables, views as well as base tables, while the VIEWS view contains information for views only.

SCHEMATA	REFERENTIAL CONSTRAINTS
DOMAINS	CHECK CONSTRAINTS
TABLES	KEY COLUMN USAGE
VIEWS	ASSERTIONS
COLUMNS	VIEW TABLE USAGE
TABLE PRIVILEGES	VIEW COLUMN USAGE
COLUMN PRIVILEGES	CONSTRAINT TABLE USAGE
USAGE PRIVILEGES	CONSTRAINT COLUMN USAGE
DOMAIN CONSTRAINTS	CONSTRAINT DOMAIN USAGE
TABLE CONSTRAINTS	

Reference [4.19] gives more details; in particular, it shows how to formulate queries against the Information Schema (the process is not as straightforward as you might expect).

4.4 VIEWS

Here is an example of an SQL view definition:

```
CREATE VIEW GOOD_SUPPLIER
AS SELECT S#, STATUS, CITY
FROM S
WHERE STATUS > 15 ;
```

And here is an example of an SQL query against this view:

```
SELECT S#, STATUS
FROM   GOOD_SUPPLIER
WHERE  CITY = 'London' ;
```

Substituting the view definition for the reference to the view name, we obtain an expression that looks something like this (note the **nested subquery** in the FROM clause):

```
SELECT GOOD_SUPPLIER.S#, GOOD_SUPPLIER.STATUS
FROM ( SELECT S#, STATUS, CITY
      FROM   S
      WHERE  STATUS > 15 ) AS GOOD_SUPPLIER
WHERE  GOOD_SUPPLIER.CITY = 'London' ;
```

And this expression can then be simplified to something like this:

```
SELECT S#, STATUS
FROM   S
WHERE  STATUS > 15
AND    CITY = 'London' ;
```

This latter is the query that is actually executed.

By way of another example, consider the following DELETE operation:

```
DELETE
FROM   GOOD_SUPPLIER
WHERE  CITY = 'London' ;
```

The DELETE that is actually executed looks something like this:

```
DELETE
FROM   S
WHERE  STATUS > 15
AND    CITY = 'London' ;
```

4.5 TRANSACTIONS

SQL's analogs of our COMMIT and ROLLBACK statements are called **COMMIT WORK** and **ROLLBACK WORK**, respectively (the keyword WORK is optional in both cases). However, SQL does not include any explicit BEGIN TRANSACTION statement. Instead, a transaction is begun implicitly whenever the program executes a “**transaction-initiating**” statement and does not already have a transaction in progress. Details of exactly which SQL statements are transaction-initiating would be out of place here; suffice it to say that all of the statements discussed in this chapter are transaction-initiating (except for COMMIT and ROLLBACK themselves, of course).

4.6 EMBEDDED SQL

Most SQL products allow SQL statements to be executed both **directly** (i.e., interactively from an online terminal) and as part of an application program (i.e., the SQL statements can be **embedded**, meaning they can be intermixed with the programming language statements

of such a program). In the embedded case, moreover, the application program can typically be written in a variety of host languages—COBOL, Java, PL/I, etc.* In this section we consider the embedded case specifically.

The fundamental principle underlying embedded SQL, which we refer to as the **dual-mode principle**, is that *any SQL statement that can be used interactively can also be used in an application program*. Of course, there are various differences of detail between a given interactive SQL statement and its embedded counterpart, and retrieval operations in particular require significantly extended treatment in a host program environment (see later in this section), but the principle is nevertheless broadly true. (Its converse is not, by the way—several embedded SQL statements cannot be used interactively, as we will see.)

Before we can discuss the actual statements of embedded SQL, it is necessary to cover a number of preliminary details. Most of those details are illustrated by the program fragment shown in Fig. 4.3. (To fix our ideas we assume the host language is PL/I. Most of the ideas translate into other host languages with only minor changes.) Points arising:

1. Embedded SQL statements are prefixed by **EXEC SQL**, to distinguish them from statements of the host language, and are terminated by a special **terminator** symbol (a semicolon for PL/I).
2. An *executable* SQL statement (for the rest of this section we will mostly drop the “embedded” qualifier) can appear wherever an executable host statement can appear. Note that “executable,” by the way: Unlike interactive SQL, embedded SQL includes some statements that are purely declarative, not executable. For example, **DECLARE CURSOR** is not an executable statement (see the subsection “Operations Involving

```
EXEC SQL BEGIN DECLARE SECTION ;

    DCL SQLSTATE CHAR(5) ;
    DCL P# CHAR(6) ;
    DCL WEIGHT FIXED DECIMAL(5,1) ;

EXEC SQL END DECLARE SECTION ;

P# = 'P2' ;
EXEC SQL SELECT P.WEIGHT          /* for example */
        INTO   :WEIGHT
        FROM   P
        WHERE  P.P# = :P# ;
IF SQLSTATE = '00000'
THEN ... ;                          /* WEIGHT = retrieved value */
ELSE ... ;                          /* some exception occurred */
```

Fig. 4.3 Fragment of a PL/I program with embedded SQL

*The SQL standard [4.22] currently supports Ada, C, COBOL, Fortran, M (previously called MUMPS), Pascal, and PL/I. Java support is not included at the time of writing, but it is due to be added soon (see reference [4.6], also Appendix B), and some products already support it.

Cursors" later), nor are **BEGIN** and **END DECLARE SECTION** (see paragraph 5 below), and nor is **WHENEVER** (see paragraph 9 below).

3. SQL statements can include references to **host variables**; such references must include a **colon prefix** to distinguish them from SQL column names. Host variables can appear in embedded SQL wherever a literal can appear in interactive SQL. They can also appear in an **INTO** clause on **SELECT** (see paragraph 4 below) or **FETCH** (again, see the subsection "Operations Involving Cursors" later) to designate targets for retrieval operations.
4. Notice the **INTO** clause on the **SELECT** statement in Fig. 4.3. The purpose of that clause is (as just indicated) to specify the target variables into which values are to be retrieved; the *i*th target variable mentioned in the **INTO** clause corresponds to the *i*th value to be retrieved, as specified by the **SELECT** clause.
5. All host variables referenced in SQL statements must be declared (DCL in PL/I) within an **embedded SQL declare section**, which is delimited by the **BEGIN** and **END DECLARE SECTION** statements.
6. Every program containing embedded SQL statements must include a host variable called **SQLSTATE**. After any SQL statement has been executed, a status code is returned to the program in that variable; in particular, a status code of 00000 means the statement executed successfully, and a value of 02000 means the statement did execute but no data was found to satisfy the request. In principle, therefore, every SQL statement in the program should be followed by a test on **SQLSTATE**, and appropriate action taken if the value is not what was expected. In practice, however, such testing can often be implicit (see paragraph 9 below).
7. Host variables must have a **data type** appropriate to the uses to which they are put. In particular, a host variable that is to be used as a target (e.g., on **SELECT**) must have a data type that is compatible with that of the expression that provides the value to be assigned to that target; likewise, a host variable that is to be used as a source (e.g., on **INSERT**) must have a data type that is compatible with that of the SQL column to which values of that source are to be assigned. Similar remarks apply to a host variable that is to be used in a comparison, or indeed in any kind of operation. For details of exactly what it means for data types to be compatible in the foregoing sense, see reference [4.22].
8. Host variables and SQL columns can have the same name.
9. As already mentioned, every SQL statement should in principle be followed by a test of the returned **SQLSTATE** value. The **WHENEVER** statement is provided to simplify this process. The **WHENEVER** statement takes the form

```
EXEC SQL WHENEVER <condition> <action> ;
```

Here *<condition>* is either **SQLERROR** or **NOT FOUND**, and *<action>* is either **CONTINUE** or a **GO TO** statement. **WHENEVER** is not an executable statement; rather, it is a directive to the SQL compiler. "**WHENEVER** *<condition>* **GO TO** *<label>*" causes the compiler to insert a statement of the form "**IF** *<condition>* **GO TO** *<label>* **END IF**" after each executable SQL statement it encounters; "**WHENEVER**

<condition> CONTINUE" causes it not to insert any such statements, the implication being that the programmer will insert such statements by hand. The two <condition>s are defined as follows:

NOT FOUND	means	no data was found — SQLSTATE = 02000 (usually)
SQLERROR	means	an error occurred — see reference [4.22] for SQL.STATE

Each WHENEVER statement the SQL compiler encounters on its sequential scan through the program text (for a particular condition) overrides the previous one it found (for that condition).

10. Note finally that—to use the terminology of Chapter 2—embedded SQL constitutes a *loose coupling* between SQL and the host language.

So much for the preliminaries. In the rest of this section we concentrate on data manipulation operations specifically. As already indicated, most of those operations can be handled in a fairly straightforward fashion (i.e., with only minor changes to their syntax). Retrieval operations require special treatment, however. The problem is that such operations retrieve many rows (in general), not just one, and host languages are generally not equipped to handle the retrieval of more than one row at a time. It is therefore necessary to provide some kind of bridge between the set-level retrieval capabilities of SQL and the row-level retrieval capabilities of the host, and **cursors** provide such a bridge. A cursor is a special kind of SQL object that applies to embedded SQL only (because interactive SQL has no need of it). It consists essentially of a kind of (*logical*) *pointer* that can be used to run through a collection of rows, pointing to each of the rows in turn and thereby providing addressability to those rows one at a time. However, we defer detailed discussion of cursors to a later subsection, and consider first those statements that have no need of them.

Operations Not Involving Cursors

The data manipulation statements that do not need cursors are as follows:

- Singleton SELECT
- INSERT
- UPDATE (except the CURRENT form—see later)
- DELETE (again, except the CURRENT form—see later)

We give examples of each of these statements in turn.

Singleton SELECT: Get status and city for the supplier whose supplier number is given by the host variable GIVENS#.

```
EXEC SQL SELECT STATUS, CITY
        INTO   :RANK, :CITY
        FROM   S
        WHERE  S# = :GIVENS# ;
```

We use the term *singleton SELECT* to mean a SELECT expression that evaluates to a table containing at most one row. In the example, if there is exactly one row in table S satisfying the condition in the WHERE clause, then the STATUS and CITY values from that row will be assigned to the host variables RANK and CITY as requested, and SQLSTATE will be set to 00000; if no S row satisfies the WHERE condition, SQLSTATE will be set to 02000; and if more than one does, the program is in error, and SQLSTATE will be set to an error code.

INSERT: Insert a new part (part number, name, and weight given by host variables P#, PNAME, PWT, respectively; color and city unknown) into table P.

```
EXEC SQL INSERT
      INTO P ( P#, PNAME, WEIGHT )
      VALUES ( :P#, :PNAME, :PWT );
```

The COLOR and CITY values for the new part will be set to the applicable **default values**. See Chapter 5, Section 5.5, for a discussion of SQL default values.

UPDATE: Increase the status of all London suppliers by the amount given by the host variable RAISE.

```
EXEC SQL UPDATE S
      SET STATUS = STATUS + :RAISE
      WHERE CITY = 'London' ;
```

If no supplier rows satisfy the WHERE condition, SQLSTATE will be set to 02000.

DELETE: Delete all shipments for suppliers whose city is given by the host variable CITY.

```
EXEC SQL DELETE
      FROM SP
      WHERE :CITY =
            ( SELECT CITY
              FROM S
              WHERE S.S# = SP.S# ) ;
```

Again SQLSTATE will be set to 02000 if no SP rows satisfy the WHERE condition. Again, note the **nested subquery** (in the WHERE clause this time).

Operations Involving Cursors

Now we turn to the question of set-level retrieval—i.e., retrieval of a set containing an arbitrary number of rows, instead of at most one row as in the singleton SELECT case. As explained earlier, what is needed here is a mechanism for accessing the rows in the set one by one, and **cursors** provide such a mechanism. The process is illustrated in outline by the example of Fig. 4.4 (overleaf), which is intended to retrieve supplier details (S#, SNAME, and STATUS) for all suppliers in the city given by the host variable Y.

Explanation: The statement "DECLARE X CURSOR ..." defines a cursor called X, with an associated *table expression* (i.e., an expression that evaluates to a table), specified by the SELECT that forms part of that DECLARE. That table expression is not evaluated at this point; DECLARE CURSOR is a purely declarative statement. The expression is eval-

```

EXEC SQL DECLARE X CURSOR FOR      /* define the cursor */
SELECT S.S#, S.SNAME, S.STATUS
FROM   S
WHERE  S.CITY = :Y
ORDER  BY S# ASC ;

EXEC SQL OPEN X ;                  /* execute the query */
DO for all S rows accessible via X ;
EXEC SQL FETCH X INTO :S#, :SNAME, :STATUS ;
                                           /* fetch next supplier */
      *****
END ;
EXEC SQL CLOSE X ;                  /* deactivate cursor X */

```

Fig. 4.4 Multi-row retrieval example

uated when the cursor is opened ("OPEN X"). The statement "FETCH X INTO ..." is then used to retrieve rows one at a time from the resulting set, assigning retrieved values to host variables in accordance with the specifications of the INTO clause in that statement. (For simplicity we have given the host variables the same names as the corresponding database columns. Notice that the SELECT in the cursor declaration does not have an INTO clause of its own.) Since there will be many rows in the result set, the FETCH will normally appear within a loop; the loop will be repeated so long as there are more rows still to come in that result set. On exit from the loop, cursor X is closed ("CLOSE X").

Now we consider cursors and cursor operations in more detail. First of all, a cursor is declared by means of a **DECLARE CURSOR** statement, which takes the general form

```

EXEC SQL DECLARE <cursor name> CURSOR
FOR <table expression> [ <ordering> ] ;

```

(we are ignoring a few optional specifications in the interests of brevity). For a complete explanation of <table expression>, see Appendix A. The optional <ordering> takes the form

```

ORDER BY <order item commalist>

```

where (a) the commalist of <order item>s must not be empty—see the paragraph immediately following—and (b) each individual <order item> consists of a column name (unqualified, please note), optionally followed by ASC (ascending) or DESC (descending), and ASC is the default.

Note: We define the term *commalist* as follows. Let <xyz> denote an arbitrary syntactic category (i.e., anything that appears on the left-hand side of some BNF production rule). Then the expression <xyz commalist> denotes a sequence of zero or more <xyz>s in which each pair of adjacent <xyz>s is separated by a comma (and possibly one or more blanks). Please note that we will make extensive use of the commalist shorthand in future syntax rules (all syntax rules, that is, not just SQL ones).

As previously stated, the DECLARE CURSOR statement is declarative, not executable; it declares a cursor with the specified name and having the specified table expres-

sion and ordering permanently associated with it. The table expression can include host variable references. A program can include any number of DECLARE CURSOR statements, each of which must (of course) be for a different cursor.

Three executable statements are provided to operate on cursors: **OPEN**, **FETCH**, and **CLOSE**.

- The statement

```
EXEC SQL OPEN <cursor name> ;
```

opens or *activates* the specified cursor (which must not currently be open). In effect, the table expression associated with the cursor is evaluated (using the current values for any host variables referenced within that expression); a set of rows is thus identified and becomes the current **active set** for the cursor. The cursor also identifies a *position* within that active set, namely the position just before the first row. (Active sets are always considered to have an ordering, so the concept of position has meaning.* The ordering is either that defined by the ORDER BY clause, or a system-determined ordering in the absence of such a clause.)

- The statement

```
EXEC SQL FETCH <cursor name>  
      INTO <host variable reference commalist> ;
```

advances the specified cursor (which must be open) to the next row in the active set and then assigns the *i*th value from that row to the *i*th host variable referenced in the INTO clause. If there is no next row when FETCH is executed, SQLSTATE is set to 02000 and no data is retrieved.

- The statement

```
EXEC SQL CLOSE <cursor name> ;
```

closes or *deactivates* the specified cursor (which must currently be open). The cursor now has no current active set. However, it can subsequently be opened again, in which case it will acquire another active set—probably not exactly the same set as before, especially if the values of any host variables referenced in the cursor declaration have changed in the meantime. Note that changing the values of those host variables while the cursor is open has no effect on the current active set.

Two further statements can include references to cursors, namely the **CURRENT** forms of **UPDATE** and **DELETE**. If a cursor, X say, is currently positioned on a particu-

*Sets *per se* do not have an ordering, of course (see Chapter 5), so an “active set” is not really a set, as such, at all. It would be better to think of it as an *ordered list* or *array* (of rows).

lar row, then it is possible to UPDATE or DELETE “the current of X,” i.e., the row on which X is positioned. For example:

```
EXEC SQL UPDATE S
      SET      STATUS = STATUS + :RAISE
      WHERE    CURRENT OF X ;
```

Note: The CURRENT forms of UPDATE and DELETE are not permitted if the table expression in the cursor declaration would define a nonupdatable view if it were part of a CREATE VIEW statement (see Chapter 9, Section 9.6).

Dynamic SQL

Dynamic SQL consists of a set of embedded SQL facilities that are intended to support the construction of generalized, online, and possibly interactive applications. (Recall from Chapter 1 that an online application is an application that supports access to the database from an online terminal.) Consider what a typical online application has to do. In outline, the steps it must go through are as follows.

1. Accept a command from the terminal.
2. Analyze that command.
3. Execute appropriate SQL statements on the database.
4. Return a message and/or results to the terminal.

If the set of commands the program can accept in Step 1 is fairly small, as in the case of (perhaps) a program handling airline reservations, then the set of possible SQL statements to be executed will probably also be small and can be “hardwired” into the program. In this case, Steps 2 and 3 above will consist simply of logic to examine the input command and then branch to the part of the program that issues the predefined SQL statement(s). On the other hand, if there can be great variability in the input, then it might not be practicable to predefine and “hardwire” SQL statements for every possible command. Instead, it is probably much more convenient to construct the necessary SQL statements dynamically, and then to compile and execute those constructed statements dynamically. The facilities of dynamic SQL are provided to assist in this process.

The two principal dynamic statements are PREPARE and EXECUTE. Their use is illustrated in the following (unrealistically simple but accurate) example.

```
DCL SQLSOURCE CHAR VARYING (65000) ;

SQLSOURCE = 'DELETE FROM SP WHERE QTY < 300' ;
EXEC SQL PREPARE SQLPREPPED FROM :SQLSOURCE ;
EXEC SQL EXECUTE SQLPREPPED ;
```

Explanation:

1. The name SQLSOURCE identifies a PL/I varying-length character string variable in which the program will somehow construct the source form (i.e., character string representation) of some SQL statement—a DELETE statement, in our particular example.

2. The name `SQLPREPPED`, by contrast, identifies an *SQL* variable, not a *PL/I* variable, that will be used (conceptually) to hold the compiled form of the *SQL* statement whose source form is given in `SQLSOURCE`. (The names `SQLSOURCE` and `SQLPREPPED` are arbitrary, of course.)
3. The *PL/I* assignment statement "`SQLSOURCE = ... ;`" assigns to `SQLSOURCE` the source form of an *SQL* `DELETE` statement. In practice, of course, the process of constructing such a source statement is likely to be much more complex—perhaps involving the input and analysis of some request from the end user, expressed in natural language or some other form more user-friendly than *SQL*.
4. The *PREPARE* statement then takes that source statement and "prepares" (i.e., compiles) it to produce an executable version, which it stores in `SQLPREPPED`.
5. Finally, the *EXECUTE* statement executes that `SQLPREPPED` version and thus causes the actual `DELETE` to occur. *SQLSTATE* information from the `DELETE` is returned exactly as if the `DELETE` had been executed directly in the normal way.

Note that since it denotes an *SQL* variable, not a *PL/I* variable, the name `SQLPREPPED` does not have a colon prefix when it is referenced in the *PREPARE* and *EXECUTE* statements. Note too that such *SQL* variables do not have to be explicitly declared.

Incidentally, the process just described is exactly what happens when *SQL* statements themselves are entered interactively. Most systems provide some kind of interactive *SQL* query processor. That query processor is in fact just a particular kind of generalized online application; it is ready to accept an extremely wide variety of input, *viz.* any valid (or invalid!) *SQL* statement. It uses the facilities of dynamic *SQL* to construct suitable *SQL* statements corresponding to its input, to compile and execute those constructed statements, and to return messages and results back to the terminal.

We conclude this subsection (and this section) with a brief mention of a more recent (1995) addition to the standard known as the **SQL Call-Level Interface** ("*SQL/CLI*," *CLI* for short). *CLI* is heavily based on Microsoft's *Open Database Connectivity* interface, *ODBC*. *CLI* permits an application written in one of the usual host languages to issue database requests by invoking certain vendor-provided *CLI routines*. Those routines, which must have been linked to the application in question, then use dynamic *SQL* to perform the requested database operations on the application's behalf. (From the *DBMS's* point of view, in other words, the *CLI routines* can be thought of as just another application.)

As you can see, *SQL/CLI* (and *ODBC* too, come to that) both address the same general problem as dynamic *SQL* does: They both allow applications to be written for which the exact *SQL* statements to be executed are not known until run time. However, they actually represent a better approach to the problem than dynamic *SQL* does. There are two principal reasons for this state of affairs:

- First, dynamic *SQL* is a *source code* standard. Any application using dynamic *SQL* thus requires the services of some kind of *SQL* compiler in order to process the operations—*PREPARE*, *EXECUTE*, etc.—prescribed by that standard. *CLI*, by contrast, merely standardizes the details of certain *routine invocations* (i.e., subroutine calls,

basically); no special compiler services are needed, only the regular services of the standard host language compiler. As a result, applications can be distributed (perhaps by third-party software vendors) in “shrink-wrapped” *object code* form.

- Second, those applications can be *DBMS-independent*; that is, CLI includes features that permit the creation (again, perhaps by third-party software vendors) of generic applications that can be used with several different DBMSs, instead of having to be specific to some particular DBMS.

Interfaces such as CLI, ODBC, and JDBC (a Java variant of ODBC) are becoming increasingly important in practice, for reasons to be discussed (in part) in Chapter 20.

4.7 SQL IS NOT PERFECT

As stated in the introduction to this chapter, SQL is very far from being the “perfect” relational language—it suffers from numerous sins of both omission and commission. Specific criticisms will be offered at appropriate points in subsequent chapters, but the overriding issue is simply that SQL fails in all too many ways to support the relational model properly. As a consequence, it is not at all clear that today’s SQL products really deserve the label “relational” at all! Indeed, so far as this writer is aware, *there is no product on the market today that supports every detail of the relational model*. This is not to say that some parts of the model are unimportant; on the contrary, *every detail* of the model is important, and important, moreover, for genuinely practical reasons. Indeed, the point cannot be stressed too strongly that the purpose of relational theory is not just “theory for its own sake”; rather, the purpose is to provide a base on which to build systems that are *100 percent practical*. But the sad fact is that the vendors have not yet really stepped up to the challenge of implementing the theory in its entirety. As a consequence, the “relational” products of today all fail, in one way or another, to deliver on the full promise of relational technology.

4.8 SUMMARY

This concludes our introduction to some of the major features of the SQL standard (“SQL/92”). We emphasized the fact that SQL is very important from a commercial perspective, though it is unfortunately somewhat deficient from a pure relational point of view.

SQL includes both a **data definition** language (DDL) component and a **data manipulation** language (DML) component. The SQL DML can operate at both the external level (on views) and the conceptual level (on base tables). Likewise, the SQL DDL can be used to define objects at the external level (views), the conceptual level (base tables), and even—in most commercial systems, though not in the standard *per se*—the internal level as well (i.e., indexes and other physical storage structures). Moreover, SQL also provides certain *data control* facilities—i.e., facilities that cannot really be classified as belonging to either the DDL or the DML. An example of such a facility is the GRANT statement, which allows users to grant *access privileges* to each other (see Chapter 16).

We showed how SQL can be used to create base tables, using the **CREATE TABLE** statement. We then gave some examples of the **SELECT**, **INSERT**, **UPDATE**, and **DELETE** statements, showing in particular how **SELECT** can be used to express the relational restrict, project, and join operations. We also briefly described the **Information Schema**, which consists of a set of prescribed views of a hypothetical “Definition Schema,” and we took a look at the SQL facilities for dealing with **views** and **transactions**.

A large part of the chapter was concerned with **embedded SQL**. The basic idea behind embedded SQL is the **dual-mode principle**, i.e., the principle that (insofar as possible) *any SQL statement that can be used interactively can also be used in an application program*. The major exception to this principle arises in connection with **multi-row retrieval operations**, which require the use of a **cursor** to bridge the gap between the set-level retrieval capabilities of SQL and the row-level retrieval capabilities of host languages such as PL/I.

Following a number of necessary (though mostly syntactic) preliminaries—including in particular a brief explanation of **SQLSTATE**—we considered those operations, namely **singleton SELECT**, **INSERT**, **UPDATE**, and **DELETE**, that have no need for cursors. Then we turned to the operations that *do* need cursors, and discussed **DECLARE CURSOR**, **OPEN**, **FETCH**, **CLOSE**, and the **CURRENT** forms of **UPDATE** and **DELETE**. (The standard refers to the **CURRENT** forms of these operators as *positioned UPDATE* and *positioned DELETE*, and uses the term *searched UPDATE* and *searched DELETE* for the non-CURRENT or “out of the blue” forms.) Finally, we gave a very brief introduction to the concept of **dynamic SQL**, describing the **PREPARE** and **EXECUTE** statements in particular, and we also mentioned the **SQL Call-Level Interface, CLI**.

EXERCISES

4.1 Fig. 4.5 (overleaf) shows some sample data values for an extended form of the suppliers and parts database called the suppliers-parts-projects database. Suppliers (S), parts (P), and projects (J) are uniquely identified by supplier number (S#), part number (P#), and project number (J#), respectively. The significance of an SPJ (shipment) row is that the specified supplier supplies the specified part to the specified project in the specified quantity (and the combination S#-P#-J# uniquely identifies such a row). Write an appropriate set of SQL definitions for this database. *Note:* This database will be used as the basis for numerous exercises in subsequent chapters.

4.2 In Section 4.2 we described the **CREATE TABLE** statement as defined by the SQL standard *per se*. Many commercial SQL products support additional options on that statement, however, typically having to do with indexes, disk space allocation, and other implementation matters (thereby undermining the objectives of physical data independence and intersystem compatibility). Investigate any SQL product that might be available to you. Do the foregoing criticisms apply to that product? Specifically, what additional **CREATE TABLE** options does that product support?

4.3 Once again, investigate any SQL product that might be available to you. Does that product support the Information Schema? If not, what *does* its catalog support look like?

4.4 Give SQL formulations for the following updates to the suppliers-parts-projects database:

- Insert a new supplier S10 into table S. The name and city are Smith and New York, respectively; the status is not yet known.

S	S#	SNAME	STATUS	CITY	SPJ	S#	P#	J#	QTY
	S1	Smith	20	London		S1	P1	J1	200
	S2	Jones	10	Paris		S1	P1	J4	700
	S3	Blake	30	Paris		S2	P3	J1	400
	S4	Clark	20	London		S2	P3	J2	200
	S5	Adams	30	Athens		S2	P3	J3	200
						S2	P3	J4	500
						S2	P3	J5	600
						S2	P3	J6	400
						S2	P3	J7	800
P	P#	PNAME	COLOR	WEIGHT		S2	P5	J2	100
	P1	Nut	Red	12.0		S3	P3	J1	200
	P2	Bolt	Green	17.0		S3	P4	J2	500
	P3	Screw	Blue	17.0		S4	P6	J3	300
	P4	Screw	Red	14.0		S4	P6	J7	300
	P5	Cam	Blue	12.0		S5	P2	J2	200
	P6	Cog	Red	19.0		S5	P2	J4	100
						S5	P5	J5	500
						S5	P5	J7	100
						S5	P6	J2	200
J	J#	JNAME	CITY			S5	P1	J4	100
	J1	Sorter	Paris			S5	P3	J4	200
	J2	Display	Rome			S5	P4	J4	800
	J3	OCR	Athens			S5	P5	J4	400
	J4	Console	Athens			S5	P6	J4	500
	J5	RAID	London						
	J6	EDS	Oslo						
	J7	Tape	London						

Fig. 4.5 The suppliers-parts-projects database (sample values)

- b. Change the color of all red parts to orange.
- c. Delete all projects for which there are no shipments.

4.5 Again using the suppliers-parts-projects database, write a program with embedded SQL statements to list all supplier rows, in supplier number order. Each supplier row should be immediately followed in the listing by all project rows for projects supplied by that supplier, in project number order.

4.6 Let tables PART and PART_STRUCTURE be defined as follows:

```
CREATE TABLE PART
( P# ... , DESCRIPTION ... ,
  PRIMARY KEY ( P# ) );

CREATE TABLE PART_STRUCTURE
( MAJOR_P# ... , MINOR_P# ... , QTY ... ,
  PRIMARY KEY ( MAJOR_P# , MINOR_P# ),
  FOREIGN KEY ( MAJOR_P# ) REFERENCES PART,
  FOREIGN KEY ( MINOR_P# ) REFERENCES PART );
```

Table PART_STRUCTURE shows which parts (MAJOR_P#) contain which other parts (MINOR_P#) as first-level components. Write an SQL program to list all component parts of a given

PART_STRUCTURE	MAJOR_P#	MINOR_P#	QTY
	P1	P2	2
	P1	P3	4
	P2	P3	1
	P2	P4	3
	P3	P5	9
	P4	P5	8
	P5	P6	3

Fig. 4.6 Table PART_STRUCTURE (sample value)

part, to all levels (the **part explosion** problem). *Note:* The sample data shown in Fig. 4.6 might help you visualize this problem. We remark that table PART_STRUCTURE shows how *bill-of-materials* data—see Chapter 1, Section 1.3, subsection “Entities and Relationships”—is typically represented in a relational system.

REFERENCES AND BIBLIOGRAPHY

- 4.1 M. M. Astrahan and R. A. Lorie: “SEQUEL-XRM: A Relational System,” Proc. ACM Pacific Regional Conf., San Francisco, Calif. (April 1975).

Describes the first prototype implementation of SEQUEL, the earliest version of SQL [4.8]. See also references [4.2–4.3], which perform an analogous function for System R.

- 4.2 M. M. Astrahan *et al.*: “System R: Relational Approach to Database Management,” *ACM TODS* 1, No. 2 (June 1976).

System R was the major prototype implementation of the SEQUEL/2 (later SQL) language [4.8]. This paper describes the architecture of System R as originally planned. See also reference [4.3].

- 4.3 M. W. Blasgen *et al.*: “System R: An Architectural Overview,” *IBM Sys. J.* 20, No. 1 (February 1981).

Describes the architecture of System R as it became by the time the system was fully implemented (compare and contrast reference [4.2]).

- 4.4 Stephen Cannan and Gerard Otten: *SQL—The Standard Handbook*. Maidenhead, UK: McGraw-Hill International (1993).

“[Our] objective . . . is to provide a reference work explaining and describing [SQL/92 as originally defined] in a much less formal and very much more readable way than the standard itself” (from the book’s introduction).

- 4.5 Joe Celko: *SQL for Smarties: Advanced SQL Programming*. San Francisco, Calif.: Morgan Kaufmann (1995).

“This is the first advanced SQL book available that provides a comprehensive presentation of the techniques necessary to support your progress from casual user of SQL to expert programmer” (from the book’s own cover).

4.6 Andrew Eisenberg and Jim Melton: "SQLJ Part 0, Now Known as SQL/OLB (Object Language Bindings)," *ACM SIGMOD Record* 27, No. 4 (December 1998). See also Gray Clossman *et al.*: "Java and Relational Databases: SQLJ," *Proc. 1998 ACM SIGMOD Int. Conf. on Management of Data*, Seattle, Wash. (June 1998).

4.7 Don Chamberlin: *Using the New DB2*. San Francisco, Calif.: Morgan Kaufmann (1996).

A readable and comprehensive description of a state-of-the-art commercial SQL product, by one of the two principal designers of the original SQL language [4.8].

Note: The book also discusses "some controversial decisions" that were made in the design of SQL—primarily (a) the decision to support nulls and (b) the decision to permit duplicate rows. "My [i.e., Chamberlin's] purpose . . . is historical rather than persuasive—I recognize that nulls and duplicates are religious issues . . . For the most part, the designers of [SQL] were practical people rather than theoreticians, and this orientation was reflected in many [design] decisions." This position is very different from that of the present writer! Nulls and duplicates are *scientific* issues, not religious ones; they are discussed, scientifically, in this book in Chapters 18 and 5, respectively. As for "practical . . . rather than [theoretical]," we categorically reject the suggestion that theory is not practical; we have already stated (in Section 4.5) our position that relational theory, at least, is very practical indeed.

4.8 Donald D. Chamberlin and Raymond F. Boyce: "SEQUEL: A Structured English Query Language," *Proc. 1974 ACM SIGMOD Workshop on Data Description, Access, and Control*, Ann Arbor, Mich. (May 1974).

The paper that first introduced the SQL language (or SEQUEL, as it was originally called; the name was subsequently changed for legal reasons).

4.9 Donald D. Chamberlin *et al.*: "SEQUEL/2: A Unified Approach to Data Definition, Manipulation, and Control," *IBM J. R&D.* 20, No. 6 (November 1976). See also the errata in *IBM J. R&D.* 21, No. 1 (January 1977).

Experience from the early prototype implementation of SEQUEL, discussed in reference [4.1] and results from the usability tests reported in reference [4.28] led to the design of a revised version of the language called SEQUEL/2. The language supported by System R [4.2–4.3] was basically SEQUEL/2 (with the conspicuous absence of the so-called "assertion" and "trigger" facilities—see Chapter 8), plus certain extensions suggested by early user experience [4.10].

4.10 Donald D. Chamberlin: "A Summary of User Experience with the SQL Data Sublanguage," *Proc. Int. Conf. on Databases*, Aberdeen, Scotland (July 1980). Also available as IBM Research Report RJ2767 (April 1980).

Discusses early user experience with System R and proposes some extensions to the SQL language in the light of that experience. Certain of those extensions—EXISTS, LIKE, PREPARE, and EXECUTE—were in fact implemented in the final version of System R. *Note:* See Chapter 7 and Appendix A for a discussion of EXISTS and LIKE, respectively.

4.11 Donald D. Chamberlin *et al.*: "Support for Repetitive Transactions and *Ad Hoc* Queries in System R," *ACM TODS* 6, No. 1 (March 1981).

Gives some measurements of System R performance in both the *ad hoc* query and "canned transaction" environments. (A "canned transaction" is a simple application that accesses only a small part of the database and is compiled prior to execution time. It corresponds to what we called a *planned request* in Chapter 2, Section 2.8.) The measurements were taken on an IBM System

370 Model 158, running System R under the VM operating system. They are described as "preliminary"; with this caveat, however, the paper shows, among other things, that (a) compilation is almost always superior to interpretation, even for *ad hoc* queries, and (b) a system like System R is capable of processing several canned transactions a second, provided appropriate indexes exist in the database.

The paper is notable because it was one of the first to give the lie to the claim, frequently heard at the time, that "relational systems will never perform." Since it was first published, of course, commercial relational products have achieved transaction rates in the hundreds and even thousands of transactions per second.

- 4.12** Donald D. Chamberlin *et al.*: "A History and Evaluation of System R," CACM 24, No. 10 (October 1981).

Describes the three principal phases of the System R project (preliminary prototype, multi-user prototype, evaluation), with emphasis on the technologies of compilation and optimization that were pioneered in System R. There is some overlap between this paper and reference [4.13].

- 4.13** Donald D. Chamberlin, Arthur M. Gilbert, and Robert A. Yost: "A History of System R and SQL / Data System," Proc. 7th Int. Conf. on Very Large Data Bases, Cannes, France (September 1981).

Discusses the lessons learned from the System R prototype and describes the evolution of that prototype into the first of IBM's DB2 product family, SQL/DS (subsequently renamed "DB2 for VM and VSE").

- 4.14** C. J. Date: "A Critique of the SQL Database Language," *ACM SIGMOD Record* 14, No. 3 (November 1984). Republished in *Relational Database: Selected Writings*. Reading, Mass.: Addison-Wesley (1986).

As noted in the body of the chapter, SQL is not perfect. This paper presents a critical analysis of a number of the language's principal shortcomings (mainly from the standpoint of formal computer languages in general rather than database languages specifically). *Note*: Certain of this paper's criticisms do not apply to SQL/92.

- 4.15** C. J. Date: "What's Wrong with SQL?," in *Relational Database Writings 1985–1989*. Reading, Mass.: Addison-Wesley (1990).

Discusses some additional shortcomings of SQL, over and above those identified in reference [4.14], under the headings "What's Wrong with SQL *per se*," "What's Wrong with the SQL Standard," and "Application Portability." *Note*: Again, certain of this paper's criticisms do not apply to SQL/92.

- 4.16** C. J. Date: "SQL Dos and Don'ts," in *Relational Database Writings 1985–1989*. Reading, Mass.: Addison-Wesley (1990).

This paper offers some practical advice on how to use SQL in such a way as (a) to avoid some of the potential pitfalls arising from the problems discussed in references [4.14–4.15] and [4.18] and (b) to realize the maximum possible benefits in terms of productivity, portability, connectivity, and so forth.

- 4.17** C. J. Date: "How We Missed the Relational Boat," in *Relational Database Writings 1991–1994*. Reading, Mass.: Addison-Wesley (1995).

A succinct summary of SQL's shortcomings with respect to its support (or lack thereof) for the structural, integrity, and manipulative aspects of the relational model.

- 4.18** C. J. Date: "Grievous Bodily Harm" (in two parts), *DBP&D* 11, No. 5 (May 1998) and 11, No. 6 (June 1998); "Fifty Ways to Quote Your Query," on the *DBP&D* website www.dbpd.com (July 1998).

SQL is an extremely redundant language, in the sense that all but the most trivial of queries can be expressed in many different ways. These papers illustrate this point and discuss some of its implications. In particular, they show that the GROUP BY clause, the HAVING clause, and range variables could all be dropped from the language with effectively no loss of functionality (and the same is also true of the "IN subquery" construct). *Note:* All of these SQL constructs are explained in Chapter 7 (Section 7.7) and/or Appendix A.

- 4.19** C. J. Date and Hugh Darwen: *A Guide to the SQL Standard* (4th edition). Reading, Mass.: Addison-Wesley (1997).

A comprehensive tutorial on SQL/92, including CLI and PSM. In particular, the book contains an appendix, Appendix D, that documents "many aspects of the standard that appear to be inadequately defined, or even incorrectly defined, at this time." *Note:* References [4.4] and [4.27] are also SQL/92 tutorials.

- 4.20** C. J. Date and Colin J. White: *A Guide to DB2* (4th edition). Reading, Mass.: Addison-Wesley (1993).

Provides an extensive and thorough overview of IBM's original DB2 product (as of 1993) and some of its companion products. DB2, like SQL/DS [4.13], was based on System R.

- 4.21** Neal Fishman: "SQL du Jour," *DBP&D* 10, No. 10 (October 1997).

A depressing survey of some of the incompatibilities to be found among SQL products that all claim to "support the SQL standard."

- 4.22** International Organization for Standardization (ISO): *Database Language SQL*, Document ISO/IEC 9075:1992. Also available as American National Standards Institute (ANSI) Document ANSI X3.135-1992.

The original ISO/ANSI SQL/92 definition (known to the cognoscenti as *ISO/IEC 9075*, or sometimes just *ISO 9075*). The original single-part document has since been expanded into an open-ended series of separate parts, under the general title *Information Technology—Database Languages—SQL*. At the time of writing, the following parts have been defined (though certainly not all completed):

- Part 1: Framework (SQL/Framework)
- Part 2: Foundation (SQL/Foundation)
- Part 3: Call-Level Interface (SQL/CLI)
- Part 4: Persistent Stored Modules (SQL/PSM)
- Part 5: Host Language Bindings (SQL/Bindings)
- Part 6: XA Specialization (SQL/Transaction)
- Part 7: Temporal (SQL/Temporal)
- Part 8: *There is no Part 8*
- Part 9: Management of External Data (SQL/MED)
- Part 10: Object Language Bindings (SQL/OLB)

The SQL3 proposals that are expected to be ratified in 1999 logically belong in Parts 1, 2, 4, and 5. Working drafts describing those proposals can be found on the World Wide Web at <http://jerry.ece.umassd.edu/isowg3/dbl/BASEdocs/public>.

Note: The point is worth mentioning that, although SQL is widely recognized as the international "relational" database standard, the standard document does not describe itself as such;

in fact, it never actually uses the term "relation" at all! (As mentioned in an earlier footnote, it does not mention the term "database" either, come to that.)

- 4.23** International Organization for Standardization (ISO): *Information Technology—Database Languages—SQL—Technical Corrigendum 2*, Document ISO/IEC 9075:1992/Cor.2:1996(E).

Contains a large number of revisions and corrections to the original version of reference [4.22]. Unfortunately, those revisions and corrections fix almost none of the problems identified in reference [4.19].

- 4.24** Raymond A. Lorie and Jean-Jacques Daudenarde: *SQL and Its Applications*. Englewood Cliffs, N.J.: Prentice-Hall (1991).

An SQL "how to" book (almost half the book consists of a detailed series of case studies involving realistic applications).

- 4.25** Raymond A. Lorie and J. F. Nilsson: "An Access Specification Language for a Relational Data Base System," *IBM J. R&D.* 23, No. 3 (May 1979).

Gives more details on one particular aspect of the System R compilation mechanism [4.11, 4.25–4.26]. For any given SQL statement, the System R optimizer generates a program in an internal language called ASL (Access Specification Language). That language serves as the interface between the optimizer and the *code generator*. (The code generator, as its name implies, converts an ASL program into machine code.) ASL consists of operators such as "scan" and "insert" on objects such as indexes and stored files. The purpose of ASL was to make the overall translation process more manageable, by breaking it down into a set of well-defined subprocesses.

- 4.26** Raymond A. Lorie and Bradford W. Wade: "The Compilation of a High-Level Data Language," IBM Research Report RJ2598 (August 1979).

System R pioneered a scheme for compiling queries ahead of run time and then automatically recompiling them if the physical database structure had changed significantly in the interim. This paper describes the System R compilation and recompilation mechanism in some detail, without however getting into questions of optimization (see reference [17.34] for information on this latter topic).

- 4.27** Jim Melton and Alan R. Simon: *Understanding the New SQL: A Complete Guide*. San Mateo, Calif.: Morgan Kaufmann (1993).

A tutorial on SQL/92 (as originally defined). Melton was the editor of the original SQL/92 specification [4.22].

- 4.28** Phyllis Reisner, Raymond F. Boyce, and Donald D. Chamberlin: "Human Factors Evaluation of Two Data Base Query Languages: SQUARE and SEQUEL," Proc. NCC 44, Anaheim, Calif. Montvale, N.J.: AFIPS Press (May 1975).

SQL's predecessor SEQUEL [4.8] was based on an earlier language called SQUARE. The two languages were fundamentally the same, in fact, but SQUARE used a rather mathematical syntax whereas SEQUEL was based on English keywords such as SELECT, FROM, WHERE, etc. The present paper reports on a set of experiments that were carried out on the usability of the two languages, using college students as subjects. A number of revisions were made to SEQUEL as a result of that work [4.9].

- 4.29** David Rozenshtein, Anatoly Abramovich, and Eugene Birger: *Optimizing Transact-SQL: Advanced Programming Techniques*. Fremont, Calif.: SQL Forum Press (1995).

Transact-SQL is the dialect of SQL supported by the Sybase and SQL Server products. This book presents a series of programming techniques for Transact-SQL based on the use of *characteris-*

tic functions (defined by the authors as “devices that allow programmers to encode conditional logic as . . . expressions within SELECT, WHERE, GROUP BY, and SET clauses”). Although expressed in terms of Transact-SQL specifically, the ideas are actually of wider applicability. *Note:* We should perhaps add that the “optimizing” mentioned in the book’s title refers not to the DBMS optimizer component, but rather to “optimizations” that can be done by users themselves, by hand.

ANSWERS TO SELECTED EXERCISES

```

4.1 CREATE TABLE S
    ( S#      CHAR(5),
      SNAME   CHAR(20),
      STATUS  NUMERIC(5),
      CITY    CHAR(15),
      PRIMARY KEY ( S# ) ) ;

CREATE TABLE P
    ( P#      CHAR(6),
      PNAME   CHAR(20),
      COLOR   CHAR(6),
      WEIGHT  NUMERIC(5,1),
      CITY    CHAR(15),
      PRIMARY KEY ( P# ) ) ;

CREATE TABLE J
    ( J#      CHAR(4),
      JNAME   CHAR(20),
      CITY    CHAR(15),
      PRIMARY KEY ( J# ) ) ;

CREATE TABLE SPJ
    ( S#      CHAR(5),
      P#      CHAR(6),
      J#      CHAR(4),
      QTY     NUMERIC(9),
      PRIMARY KEY ( S#, P#, J# ),
      FOREIGN KEY ( S# ) REFERENCES S,
      FOREIGN KEY ( P# ) REFERENCES P,
      FOREIGN KEY ( J# ) REFERENCES J ) ;

4.4 a. INSERT INTO S ( S#, SNAME, CITY )
      VALUES ( 'S10', 'Smith', 'New York' ) ;

      STATUS here is set to the applicable default value.

b. UPDATE P
   SET   COLOR = 'Orange'
  WHERE COLOR = 'Red' ;

c. DELETE
   FROM   J
  WHERE  J# NOT IN
        ( SELECT J#
          FROM   SPJ ) ;

```

Note the nested subquery and the IN operator (actually, the *negated* IN operator) in the solution to part c. here. See Chapter 7 for further explanation.

4.5 Note that there might be some suppliers who supply no projects at all; the following solution deals with such suppliers satisfactorily (how, exactly?). First we define two cursors, CS and CJ, as follows:

```
EXEC SQL DECLARE CS CURSOR FOR
  SELECT S.S#, S.SNAME, S.STATUS, S.CITY
  FROM S
  ORDER BY S# ;

EXEC SQL DECLARE CJ CURSOR FOR
  SELECT J.J#, J.JNAME, J.CITY
  FROM J
  WHERE J.J# IN
    ( SELECT SPJ.J#
      FROM SPJ
      WHERE SPJ.S# = :CS_S# )
  ORDER BY J# ;
```

(Note the nested subquery and the IN operator once again.)

When cursor CJ is opened, host variable CS_S# will contain a supplier number value, fetched via cursor CS. The procedural logic is essentially as follows:

```
EXEC SQL OPEN CS ;
DO for all S rows accessible via CS ;
  EXEC SQL FETCH CS INTO :CS_S#, :CS_SN, :CS_ST, :CS_SC ;
  print CS_S#, CS_SN, CS_ST, CS_SC ;
  EXEC SQL OPEN CJ ;
  DO for all J rows accessible via CJ ;
    EXEC SQL FETCH CJ INTO :CJ_J#, :CJ_JN, :CJ_JC ;
    print CJ_J#, CJ_JN, CJ_JC ;
  END DO ;
  EXEC SQL CLOSE CJ ;
END DO ;
EXEC SQL CLOSE CS ;
```

4.6 This is a good example of a problem that SQL/92 does not handle well. The basic difficulty is as follows: We need to "explode" the given part to n levels, where the value of n is unknown at the time of writing the program. A comparatively straightforward way of performing such an n -level "explosion"—if it were possible—would be by means of a recursive program, in which each recursive invocation creates a new cursor, as follows:

```
CALL RECURSION ( GIVEN_P# ) ;

RECURSION: PROC ( UPPER_P# ) RECURSIVE ;
  DCL UPPER_P# ... ;
  DCL LOWER_P# ... ;
  EXEC SQL DECLARE C "reopenable" CURSOR FOR
    SELECT MINOR_P#
    FROM PART_STRUCTURE
    WHERE MAJOR_P# = :UPPER_P# ;

  print UPPER_P# ;
  EXEC SQL OPEN C ;
  DO for all PART_STRUCTURE rows accessible via C ;
    EXEC SQL FETCH C INTO :LOWER_P# ;
    CALL RECURSION ( LOWER_P# ) ;
  END DO ;
  EXEC SQL CLOSE C ;
END PROC ;
```

We have assumed here that the (fictitious) specification "reopenable" on DECLARE CURSOR means that it is legal to OPEN that cursor even if it is already open, and that the effect of such an OPEN is to create a new *instance* of the cursor for the specified table expression (using the current values of any host variables referenced in that expression). We have assumed further that references to such a cursor in FETCH (etc.) are references to the "current" instance, and that CLOSE destroys that instance and reinstates the previous instance as "current." In other words, we have assumed that a reopenable cursor forms a *stack*, with OPEN and CLOSE serving as the "push" and "pop" operators for that stack.

Unfortunately, these assumptions are purely hypothetical today. There is no such thing as a reopenable cursor in SQL today (indeed, an attempt to OPEN a cursor that is already open will fail). The foregoing code is illegal. But the example makes it clear that "reopenable cursors" would be a very desirable extension.

Since the foregoing approach does not work, we give a sketch of a possible (but very inefficient) approach that does.

```
CALL RECURSION ( GIVENP# ) ;
RECURSION: PROC ( UPPER_P# ) RECURSIVE ;
  DCL UPPER_P# ... ;
  DCL LOWER_P# ... INITIAL ( ' ' ) ;
  EXEC SQL DECLARE
    C CURSOR FOR
    SELECT MINOR_P#
    FROM PART_STRUCTURE
    WHERE MAJOR_P# = :UPPER_P#
    AND MINOR_P# > :LOWER_P#
    ORDER BY MINOR_P# ;

  print UPPER_P# ;
  DO "forever" ;
    EXEC SQL OPEN C ;
    EXEC SQL FETCH C INTO :LOWER_P# ;
    EXEC SQL CLOSE C ;
    IF no "lower_P#" retrieved THEN RETURN ; END IF ;
    IF "lower_P#" retrieved THEN
      CALL RECURSION ( LOWER_P# ) ; END IF ;
  END DO ;
END PROC ;
```

Observe in this solution that the same cursor is used on every invocation of RECURSION. (By contrast, new instances of UPPER_P# and LOWER_P# are created dynamically each time RECURSION is invoked; those instances are destroyed at completion of that invocation.) Because of this fact, we have to use a trick—

```
... AND MINOR_P# > :LOWER_P# ORDER BY MINOR_P#
```

—so that, on each invocation of RECURSION, we ignore all immediate components (LOWER_P#s) of the current UPPER_P# that have already been processed.

See (a) references [4.5] and [4.7] for a discussion of some alternative SQL-style approaches to this problem, (b) Chapter 6 (end of Section 6.7) for a description of a pertinent relational operator called *transitive closure*, and (c) Appendix B for an overview of some relevant SQL3 facilities.