21.7 Philip A. Bernstein, David W. Shipman, and James B. Rothnie, Jr. "Concurrency Control in a System for Distributed Databases (SDD-1)." *ACM TODS 5*, No. 1 (March 1980).

See the annotation to reference [21.26].

21.8 Yuri Breitbart, Hector Garcia-Molina, and Avi Silberschatz. "Overview of Multidatabase Transaction Management." *The VLDB Journal 1*, No. 2 (October 1992).

The term **multidatabase system** is sometimes used to mean a distributed system—usually *heterogeneous*—with full local autonomy. (The term "federated system" is also sometimes used with much the same meaning.) In such a system, purely local transactions are managed entirely by the local DBMSs, but global transactions are a different matter. This paper provides a survey of recent results and current research in this latter ("increasingly important") field.

21.9 Stefano Ceri and Giuseppe Pelagatti. *Distributed Databases: Principles and Systems*. New York, N.Y.: McGraw-Hill (1984).

21.10 E. F. Codd. "Is Your DBMS Really Relational?" *Computerworld* (October 14th, 1985); "Does Your DBMS Run by the Rules?" *Computerworld* (October 21st, 1985).

21.11 D. Daniels *et al.* "An Introduction to Distributed Query Compilation in R*." In *Distributed Data Bases* (ed., H.-J. Schneider): Proc. 2nd International Symposium on Distributed Data Bases (September 1982). New York, N.Y.: North-Holland (1982).

See the annotation to reference [21.30].

21.12 C. J. Date. "Distributed Databases." Chapter 7 of C. J. Date, *An Introduction to Database Systems: Volume II*. Reading, Mass.: Addison-Wesley (1983).

Portions of the present chapter are based on this earlier publication.

21.13 C. J. Date. "What Is a Distributed Database System?" In C. J. Date, *Relational Database Writings 1985–1989*. Reading, Mass.: Addison-Wesley (1990).

The paper that introduced the "twelve objectives" for distributed systems (Section 21.3 is modeled fairly directly on this paper). As mentioned in the body of the chapter, the objective of *local autonomy* is not 100 percent achievable; there are certain situations that necessarily involve compromising on that objective somewhat. We summarize those situations below for purposes of reference.

- Individual fragments of a fragmented relation cannot normally be accessed directly, not even from the site at which they are stored.
- Individual copies of a replicated relation (or fragment) cannot normally be accessed directly, not even from the site at which they are stored.
- Let P be the primary copy of some replicated relation (or fragment) R, and let P be stored at site X. Then every site that accesses R is dependent on site X, even if another copy of R is in fact stored at the site in question.
- A relation that participates in a multi-site integrity constraint cannot be accessed for update purposes within the local context of the site at which it is stored, but only within the context of the distributed database in which the constraint is defined.
- A site that is acting as a participant in a two-phase commit process must abide by the decision (i.e., commit or rollback) of the corresponding coordinator site.

21.14 C. J. Date. "Distributed Database: A Closer Look." In C. J. Date and Hugh Darwen, *Relational Database Writings 1989–1991*. Reading, Mass.: Addison-Wesley (1992).

A sequel to reference [21.13], discussing most of the twelve objectives in considerably more depth (albeit still in tutorial style).

21.15 C. J. Date. "Why Is It So Difficult to Provide a Relational Interface to IMS?" In C. J. Date, *Relational Database: Selected Writings*. Reading, Mass.: Addison-Wesley (1986).

21.16 R. Epstein, M. Stonebraker, and E. Wong. "Distributed Query Processing in a Relational Data Base System." Proc. 1978 ACM SIGMOD International Conference on Management of Data, Austin, Texas (May/June 1978).

See the annotation to reference [21.28].

21.17 John Grant, Witold Litwin, Nick Roussopoulos, and Timos Sellis. "Query Languages for Relational Multidatabases." *The VLDB Journal 2*, No. 2 (April 1993).

Proposes extensions to the relational algebra and relational calculus for dealing with multidatabase systems [21.8]. Issues of optimization are discussed, and it is shown that every multirelational algebraic expression has a multirelational calculus equivalent ("the converse of this theorem is an interesting research problem").

21.18 J. N. Gray. "A Discussion of Distributed Systems." Proc. Congresso AICA 79, Bari, Italy (October 1979). Also available as IBM Research Report RJ2699 (September 1979).

A sketchy but good overview and tutorial.

21.19 Richard D. Hackathorn. "Interoperability: DRDA or RDA?" *InfoDB 6*, No. 2 (Fall 1991).

21.20 Michael Hammer and David Shipman. "Reliability Mechanisms for SDD-1: A System for Distributed Databases." *ACM TODS 5*, No. 4 (December 1980).

See the annotation to reference [21.26].

21.21 B. G. Lindsay *et al.* "Notes on Distributed Databases." IBM Research Report RJ2571 (July 1979).

This paper (by some of the original members of the R* team) is divided into five chapters:

1. Replicated data
2. Authorization and views
3. Introduction to distributed transaction management
4. Recovery facilities
5. Transaction initiation, migration, and termination

Chapter 1 discusses the update propagation problem. Chapter 2 is almost totally concerned with authorization in a *non*distributed system (in the style of System R), except for a few remarks at the end. Chapter 3 considers transaction initiation and termination, concurrency control, and recovery control, all rather briefly. Chapter 4 is devoted to the topic of recovery in the *non*distributed case (again). Finally, Chapter 5 discusses distributed transaction management in some detail; in particular, it gives a very careful presentation of two-phase commit.

21.22 C. Mohan and B. G. Lindsay. "Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions." Proc. 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (1983).

See the annotation to reference [21.30].

21.23 Scott Newman and Jim Gray. "Which Way to Remote SQL?" *Database Programming & Design 4*, No. 12 (December 1991).

21.24 M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Englewood Cliffs, N. J.: Prentice-Hall (1991).

21.25 James B. Rothnie, Jr., and Nathan Goodman. "A Survey of Research and Development in Distributed Database Management." Proc. 3rd International Conference on Very Large Data Bases, Tokyo, Japan (October 1977). Also published in reference [21.5].

A very useful early survey. The field is discussed under the following headings:

1. Synchronizing update transactions
2. Distributed query processing
3. Handling component failures
4. Directory management
5. Database design

The last of these refers to the *physical* design problem—what we called the *allocation* problem in Section 21.8.

21.26 J. B. Rothnie, Jr., *et al.* "Introduction to a System for Distributed Databases (SDD-1)." *ACM TODS 5*, No. 1 (March 1980).

References [21.6–21.7], [21.20], [21.26], and [21.31] are all concerned with the early distrib-

uted prototype SDD-1, which ran on a collection of DEC PDP-10s interconnected via the ARPANET. It provided full location, fragmentation, and replication independence. We offer a few comments here on selected aspects of the system.

Query processing: The SDD-1 query optimizer (see references [21.6] and [21.31]) made extensive use of an operator called **semijoin,** which we explain as follows. (*Note:* The term "semijoin" first appeared with a somewhat different meaning in a paper by Palermo [18.15].) Given two relations *A* and *B*, the expression "*A* semijoin *B*" is defined to be equal to the join of *A* and *B*, projected back on to the attributes of *A*. In other words, the semijoin operation yields that subset of the tuples of *A* that match at least one tuple in *B* under the joining condition. For example, the semijoin of the supplier relation S and the shipment relation SP (over supplier numbers) is the set of S tuples for suppliers who supply at least one part—namely, the set of S tuples for suppliers S1, S2, S3, and S4, given our usual sample data values. Note that the expressions "*A* semijoin *B*" and "*B* semijoin *A*" are not equivalent, in general.

The advantage of using semijoins in distributed query processing is that they can have the effect of reducing the volume of data shipped across the network. For example, suppose that the supplier relation S is stored at site A and the shipment relation SP is stored at site B, and the query is "Compute the join of S and SP over S#." Instead of shipping the entire relation S to B (say), we can do the following:

- Compute the projection (TEMP1) of SP over S# at B.
- Ship TEMP1 to A.
- Compute the semijoin (TEMP2) of TEMP1 and S over S# at A.
- Ship TEMP2 to B.
- Compute the semijoin of TEMP2 and SP over S# at B. The result is the answer to the original query.

This procedure will obviously reduce the total amount of data movement across the network if and only if

```
size (TEMP1)  +  size (TEMP2)  <  size (S)
```

where the *size* of a relation is the cardinality of that relation multiplied by the width of an individual tuple (in bits, say). The optimizer thus clearly needs to be able to estimate the size of intermediate results such as TEMP1 and TEMP2.

Update propagation: The SDD-1 update propagation algorithm is "propagate immediately" (there is no notion of a primary copy).

Concurrency control: Concurrency control is based on a technique called **timestamping,** instead of on locking; the objective is to avoid the message overhead associated with locking, but the price seems to be that there is not in fact very much concurrency! The details are beyond the scope of this book (though the annotation to reference [14.2] does describe the basic idea very briefly); see reference [21.7] or reference [21.12] for more information.

Recovery control: Recovery is based on a *four*-phase commit protocol; the intent is to make the process more resilient than the conventional two-phase commit protocol to a failure at the Coordinator site, but unfortunately it also makes the process considerably more complex. The details are (again) beyond the scope of this book.

Catalog: The catalog is managed by treating it as if it were ordinary user data—it can be arbitrarily fragmented, and the fragments can be arbitrarily replicated and distributed, just like any other data. The advantages of this approach are obvious. The disadvantage is, of course, that since the system has no *a priori* knowledge of the location of any given piece of the catalog, it is necessary to maintain a higher-level catalog—the **directory locator**—to provide exactly that information! The directory locator is fully replicated (i.e., a copy is stored at every site).

21.27 P. G. Selinger and M. E. Adiba. "Access Path Selection in Distributed Data Base Management Systems." In S. M. Deen and P. Hammersley (eds.), Proc. International Conference on Data Bases, Aberdeen, Scotland (July 1980). London, England: Heyden and Sons Ltd. (1980).

See the annotation to reference [21.30].

21.28 M. R. Stonebraker and E. J. Neuhold. "A Distributed Data Base Version of INGRES." Proc. 2nd Berkeley Conference on Distributed Data Management and Computer Networks, Lawrence Berkeley Laboratory (May 1977).

References [21.16] and [21.28–21.29] are all concerned with the distributed INGRES prototype. Distributed INGRES consists of several copies of University INGRES, running on several interconnected DEC PDP-11s. It supports location independence (like SDD-1 and R*); it also supports data fragmentation (via restriction but not projection), with fragmentation independence, and data replication for such fragments, with replication independence. Unlike SDD-1 and R*, Distributed INGRES does not necessarily assume that the communication network is slow; on the contrary, it is designed to handle both "slow" (long-haul) networks and local (i.e., comparatively fast) networks. The query optimizer understands the difference between the two cases. The query optimization algorithm is basically an extension of the INGRES decomposition strategy described in Chapter 18 of this book; it is described in detail in reference [21.16].

Distributed INGRES provides two update propagation algorithms: a "performance" algorithm, which works by updating a primary copy and then returning control to the transaction (leaving the propagated updates to be performed in parallel by a set of slave processes); and a "reliable" algorithm, which updates all copies immediately (see reference [21.29]). Concurrency control is based on locking in both cases. Recovery is based on two-phase commit (with certain improvements that are beyond the scope of this book, as in the case of R*; again, see reference [21.29] or reference [21.14]).

As for the catalog, Distributed INGRES uses a combination of full replication for certain portions of the catalog—basically the portions containing a logical description of the relations visible to the user and a description of how those relations are fragmented—together with purely local catalog entries for other portions, such as the portions describing local physical storage structures, local database statistics (used by the optimizer), and security and integrity constraints.

21.29 M. R. Stonebraker. "Concurrency Control and Consistency of Multiple Copies in Distributed INGRES." *IEEE Transactions on Software Engineering SE-5*, No. 3 (May 1979).

See the annotation to reference [21.28].

21.30 R. Williams *et al.* "R*: An Overview of the Architecture." In P. Scheuermann (ed.), *Improving Database Usability and Responsiveness*. New York, N.Y.: Academic Press (1982). Also available as IBM Research Report RJ3325 (December 1981).

References [21.11], [21.22], [21.27], and [21.30] are all concerned with R*, the distributed version of the original System R prototype. R* provides location independence, but no fragmentation and no replication, and therefore no fragmentation or replication independence either. The question of update propagation does not arise, for the same reason. Concurrency control is based on locking (note that there is only one copy of any object to be locked; the question of a primary copy also does not arise). Recovery is based on two-phase commit, but with certain improvements that have the effect of reducing the number of messages required (see reference [21.22] or reference [21.14]).

21.31 Eugene Wong. "Retrieving Dispersed Data from SDD-1: A System for Distributed Databases." In reference [21.5].

See the annotation to reference [21.26].

21.32 C. T. Yu and C. C. Chang. "Distributed Query Processing." *ACM Comp. Surv. 16*, No. 4 (December 1984).

A tutorial survey of techniques for query optimization in distributed systems. Includes an extensive bibliography.

PART VI

OBJECT-ORIENTED SYSTEMS

Object-oriented technology is an important new area within the overall field of database management. Unfortunately, it is also an area in which there is a certain amount of confusion, as the reader will soon see. Part of the problem is that there is no universally accepted, abstract, formally defined "object data model" (contrast the situation with relational technology): Rather, the term **OO**—which we use generically as an abbreviation for "object-oriented" or "object orientation," as the context demands—is best thought of as a convenient label for a collection of interconnected ideas. Of those ideas, moreover, some are tightly interwoven and some are not; some are essential and some are not; some are good and some are not; some are new and some are not (very few are truly new, as a matter of fact); some really do apply to databases and some do not; and so on.

Regarding this latter point, incidentally, another source of confusion is the fact that the OO label is applied to a variety of quite distinct disciplines. For example, it is used to describe a certain *graphic interface* style, a certain *programming* style, a certain set of programming *languages* (not the same thing as programming style!), a certain set of *analysis and design* methodologies, and so on, as well as a certain approach to database management *per se*. And just because OO might be good for some of these problems, it does not follow that it is necessarily good for others.

In the chapters that follow, we attempt to clarify these matters. Specifically, we present a tutorial on OO as it applies to database systems, an analysis (not uncritical) of OO database concepts, and a proposal for a rapprochement between OO and relational database technology. *Note:* We assume throughout that the reader has a reasonable prior understanding of conventional (relational) database technology.

22 Basic OO Concepts

22.1 Introduction

"Everyone knows" that today's relational products are inadequate in a variety of ways. And some people—not this writer!—would argue that the relational model is inadequate too. Be that as it may, some of the new features that seem to be needed in DBMSs have existed for many years in **object-oriented programming languages;** it is thus natural to investigate the idea of incorporating those features into database systems, and hence to consider the possibility of **object-oriented database systems**. And, of course, many researchers, and some vendors, have been doing exactly this over the past few years.

To repeat, OO database systems have their origins in OO programming languages. And the basic idea in both disciplines is the same—to wit: Users should not have to wrestle with computer-oriented constructs such as bits and bytes (or even records and fields), but rather should be able to deal with **objects,** and **operations** on those objects, that more closely resemble their counterparts in the real world. For example, instead of having to think in terms of a "DEPT tuple" plus a collection of corresponding "EMP tuples" that include "foreign key values" that "reference" the "primary key value" in that "DEPT tuple," the user should be able to think directly of a *department object* that actually contains a corresponding set of *employee objects*. And instead of, e.g., having to "insert" a "tuple" into the "EMP relation" with an appropriate "foreign key value" that "references" the "primary key value" of some "tuple" in the "DEPT relation," the user should be able to *hire* an *employee* object directly into the relevant *department* object. In other words, the fundamental idea is to **raise the level of abstraction**.

Now, raising the level of abstraction is unquestionably a desirable goal, and the OO paradigm has been very successful in meeting that goal in the programming languages arena [22.3]. It is natural to ask, therefore, whether the same paradigm can be applied in the database arena also. Indeed, the idea of dealing with a database that is made up of **encapsulated objects** (e.g., department objects that "know what it means" to hire an employee or change their manager or cut their budget), instead of having to understand relations, tuple updates, foreign keys, etc., is naturally much more attractive from the user's point of view—at least at first sight.

A word of caution is appropriate, however. The point is, although the programming language and database management disciplines certainly do have a lot in common, they also differ in certain important respects. In particular, an application program (by definition) is intended to solve some specific problem, whereas a database (by definition) is intended to solve a variety of different problems, some of which might not even be known at the time the database is established. In the application programming environment, therefore, embedding a lot of "intelligence" into encapsulated objects is clearly a good idea: It reduces the amount of manipulative code that has to be written to use those objects, it improves programmer productivity, it increases application maintainability, and so on. In the database environment, by contrast, embedding a lot of intelligence into the database might or might not be a good idea: It might simplify some problems, but it might at the same time make others more difficult or even impossible.

(Incidentally, the foregoing point is exactly one of the arguments against prerelational database systems such as IMS. A department object that contains a set of employee objects is conceptually very similar to an IMS *hierarchy* in which department "segments"—this is the IMS term—are superior to employee "segments." Such a hierarchy is well suited to problems like "Find employees that work in the accounting department." It is not well suited to problems like "Find departments that employ MBAs." Thus, many of the arguments that were made against the hierarchic approach in the 1970s are surfacing again in the OO context.)

The foregoing caveat notwithstanding, many people believe that OO systems are the next big step forward in database technology. In particular, many believe that OO techniques are the approach of choice for new application areas such as

- **computer-aided design and manufacturing** (CAD/CAM)
- **computer-integrated manufacturing** (CIM)
- **computer-aided software engineering** (CASE)
- **geographic information systems** (GIS)
- **science and medicine**
- **document storage and retrieval**

and so forth (note that all of the above represent areas in which today's relational products do tend to run into trouble). There have certainly been numerous technical papers on such matters in the literature over the past few years, and more recently several commercial products have begun to appear. In this part of the book, therefore, we will take a look at what OO technology is all about.

The aim of this first chapter is simply to introduce the most important concepts of the OO approach, and in particular to describe those concepts **from a database perspective** (much of the literature, by contrast, presents the ideas very much from a *programming* perspective instead). The structure of the chapter is as follows. Following this introductory section, Section 22.2 presents a motivating example—an example, that is, that today's relational products do *not* handle satisfactorily, and hence one that OO technology stands a good chance of doing better on. Next, Sec-

tion 22.3 presents an overview of *objects, object classes, messages,* and *methods;* Sections 22.4–22.6 then focus in on certain specific aspects of these concepts and discuss them in more depth. Section 22.7 briefly considers *class hierarchies;* Section 22.8 discusses a few miscellaneous topics, and Section 22.9 presents a summary.

One last preliminary remark: Despite the fact that OO systems were originally meant specifically for "complex" applications such as CAD/CAM, CIM, CASE, etc., we deliberately use only very simple examples in our discussions, for reasons of space and familiarity. Of course, adopting this simplifying approach in no way invalidates the presentation. In any case, OO technology is intended to be suitable for "simple" applications too.

22.2 A Motivating Example

We begin with a simple example, due to Stonebraker [19.10] and elaborated by the present author in reference [22.5], that illustrates some of the problems with today's relational products. The database (which might be thought of as a grossly simplified approximation to a CAD/CAM database) concerns *rectangles,* all of which are assumed for simplicity to be "square on" to the X and Y axes—i.e., all sides are either vertical or horizontal. Each rectangle can thus be uniquely represented by the coordinates (*x1,y1*) and (*x2,y2*), respectively, of its bottom left and top right corners (see Fig. 22.1). In SQL:

```
CREATE TABLE RECTANGLES
     ( RECTID ... , X1 ... , X2 ... , Y1 ... , Y2 ... , ... ,
       UNIQUE ( RECTID ) ,
       UNIQUE ( X1, X2, Y1, Y2 ) ) ;
```

Now consider the query "Find all rectangles that overlap the unit square (0,1,0,1)" (see Fig. 22.2). The "obvious" representation of this query in SQL is:

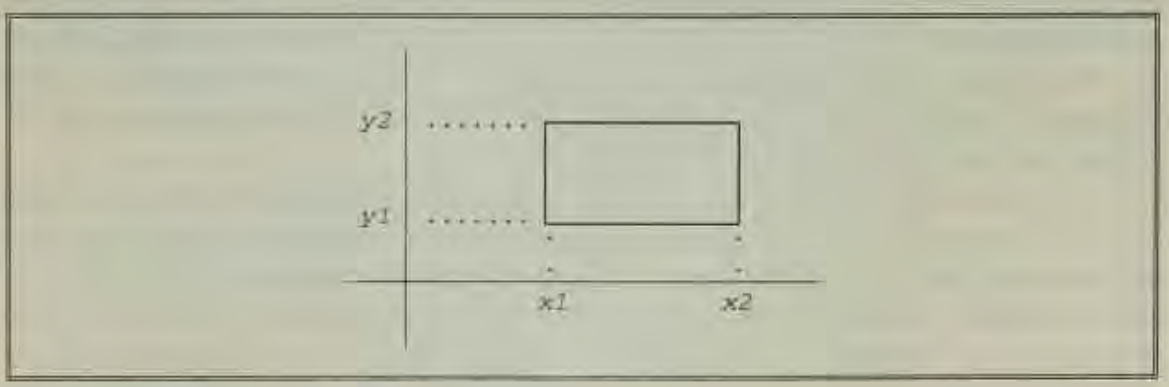

FIG. 22.1 The rectangle (*x1, x2, y1, y2*)

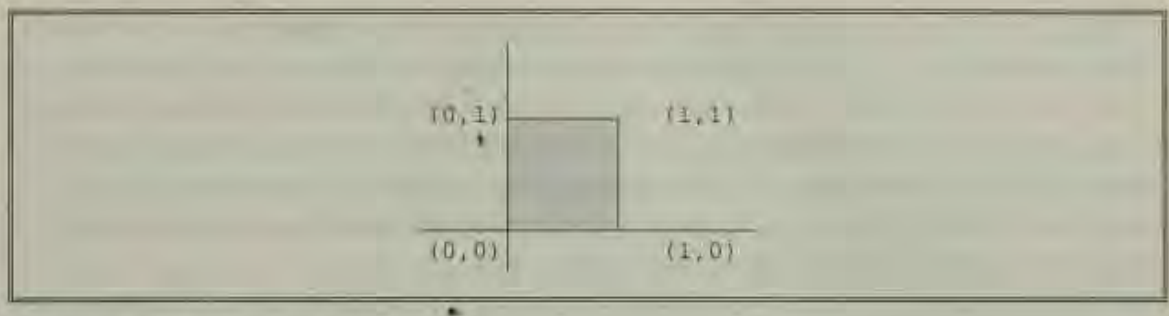

FIG. 22.2 The unit square (0,1,0,1)

```
SELECT ...
FROM   RECTANGLES
WHERE  ( X1 >= 0 AND X1 <= 1 AND Y1 >= 0 AND Y1 <= 1 )
                        - bottom left corner inside unit square
OR     ( X2 >= 0 AND X2 <= 1 AND Y2 >= 0 AND Y2 <= 1 )
                        - top right corner inside unit square
OR     ( X1 >= 0 AND X1 <= 1 AND Y2 >= 0 AND Y2 <= 1 )
                        - top left corner inside unit square
OR     ( X2 >= 0 AND X2 <= 1 AND Y1 >= 0 AND Y1 <= 1 )
                        - bottom right corner inside unit square
OR     ( X1 <= 0 AND X2 >= 1 AND Y1 <= 0 AND Y2 >= 1 )
                        - rectangle totally includes unit square
OR     ( X1 <= 0 AND X2 >= 1 AND Y1 >= 0 AND Y1 <= 1 )
                        - bottom edge crosses unit square
OR     ( X1 >= 0 AND X1 <= 1 AND Y1 <= 0 AND Y2 >= 1 )
                        - left edge crosses unit square
OR     ( X2 >= 0 AND X2 <= 1 AND Y1 <= 0 AND Y2 >= 1 )
                        - right edge crosses unit square
OR     ( X1 <= 0 AND X2 >= 1 AND Y2 >= 0 AND Y2 <= 1 )
                        - top edge crosses unit square
;
```

Readers should take the time to convince themselves that this formulation is indeed correct before continuing. Diagrams will probably prove helpful in this regard.

With a little further thought, however, it can be seen that the query can be expressed somewhat more simply as:

```
SELECT ...
FROM   RECTANGLES
WHERE  ( X1 <= 1 AND Y1 <= 1
                  - bottom left corner is "downwind" of (1,1)
AND      X2 >= 0 AND Y2 >= 0 )
                  - top right corner is "upwind" of (0,0)
;
```

Again, diagrams will probably prove helpful in understanding this formulation.

The question now is: Could the system optimizer transform the original long form of the query into the corresponding short form? In other words, suppose the user expresses the query in the "obvious" (and obviously inefficient) long form; would the system be capable of reducing that query to the more efficient short form before exe-

cuting it? Reference [22.5] gives evidence to suggest that the answer to this question is almost certainly *no*, at least so far as today's commercial optimizers are concerned.

In any case, despite the fact that we just referred to the short form as "more efficient," it is likely that performance on the short form will still be unacceptably poor in most of today's relational products, given the usual storage structures—typically B-trees—supported by those products (on average, the system will examine 50 percent of the index entries for each of X1, X2, Y1, and Y2).

Thus we see that today's relational products are indeed inadequate in certain respects. To be specific, problems like the rectangles problem show clearly that certain "simple" user requests

1. are unreasonably difficult to express, and
2. execute with unacceptably poor performance,

in those products. Such considerations provide much of the motivation behind the current interest in OO systems.

Note: We will give a "good" solution to the rectangles problem in Chapter 25 (Section 25.3).

22.3 Objects, Methods, and Messages

In this section, we introduce some of the principal terms and concepts of the OO approach, namely *object* itself (of course), *object class*, *method*, and *message*. We will also relate these notions to more familiar terms and concepts wherever possible or appropriate. In fact, it is probably helpful to show a rough mapping of the OO terms to traditional programming terms right at the outset (refer to Fig. 22.3).

Caveat: Before we start getting into details, it is as well to warn readers not to expect the kind of precision they are accustomed to in the relational world. Indeed, many OO concepts—or the published definitions of those concepts, at any rate—are quite imprecise, and there is very little true consensus and much disagreement, even at the most basic level (see references [22.2], [22.10–22.11], [22.14], and [22.16]). In particular, there is no abstract, formally defined "object data model," nor is there even consensus on an *in*formal model. (For such reasons we will always place phrases such

OO term	Programming term
object	variable (if "mutable") or value (if "immutable")
object class	data type
method	function
message	call

FIG. 22.3 OO terminology

as "the OO model" in quotes in this book.) In fact, there seems, rather surprisingly, to be much confusion over *levels of abstraction*—specifically, over the distinction between the model *per se* and its implementation.

The reader should also be warned that, as a consequence of the foregoing state of affairs, the definitions and explanations offered in this part of the book are *not* universally agreed upon and do *not* necessarily correspond to the way that any given OO system actually works. Indeed, just about every one of those definitions and explanations could well be challenged by some other worker in this field, and probably will be.

An Overview of OO Technology

Question: What is an object?
Answer: Everything!

It is a basic tenet of the OO approach that **"everything is an object"** (sometimes "everything is a **first-class** object"). Some objects are builtin, primitive, and **immutable;** examples might be integers (e.g., 3, 42) and character strings (e.g., "Mozart", "Hayduke Lives!"). In traditional terminology, such objects correspond to simple *values*. Other objects—typically user-created and typically **mutable**—are more complex; examples might be EMPs, DEPTs, VEHICLEs, etc. These more complex objects correspond to *variables,** of arbitrary internal complexity (where the phrase "arbitrary internal complexity" is meant to imply that such objects can make use internally of any or all of the usual programming language data types and type constructors—numbers, strings, lists, arrays, stacks, etc.). *Note:* In some systems the term *object* is used to mean the mutable case only (the term *value* then being used for the immutable case). Even in those systems where the term "object" does strictly refer to both cases, the reader should be aware that it is common in informal contexts to take the term to mean a mutable object specifically, barring explicit statements to the contrary.

Every object has a *type* (the OO term is **class**). Individual objects are sometimes referred to as object **instances** specifically, in order to distinguish them clearly from the corresponding object type or class. Please note also that the term *type* here is used with its usual programming language meaning; that is, an intrinsic aspect of any given type consists of the set of *operators* or *functions* (the OO term is **methods**) that can be applied to objects of that type (see the discussion of *encapsulation* below). We will normally assume that the type of an object is fixed at object creation time, although as we will see in Section 22.7—when we discuss class hierarchies—an object might be of several types simultaneously, and might acquire and lose types dynamically.

As an aside, we remark that some OO systems support both types and classes and therefore make a distinction between the two concepts. We will discuss this possibility further in Section 22.6.

All objects are **encapsulated**. What this means is that the representation—i.e., the

---

* Note, however, that the (unqualified) term *variable* in OO systems refers specifically to a *program* variable that is used to hold an *object ID* (see later).

internal structure—of a given object, say a given DEPT, is not visible to users of that object; instead, users know only that the object is capable of performing certain functions ("methods"). For example, the methods that apply to DEPT objects might be HIRE_EMP, FIRE_EMP, CUT_BUDGET, etc. *Note carefully that such methods constitute the ONLY operations that can be applied to the objects in question.* The code that implements those methods *is* permitted to see the internal representation of the objects, of course; to use the jargon, those methods—and only those methods—are allowed to "break encapsulation."

The advantage of encapsulation is that it allows the internal representation of objects to be changed without requiring any of the applications that use those objects to be rewritten—provided, of course, that any such change in internal representation is accompanied by a corresponding change to the code that implements the applicable methods. In other words, encapsulation implies **data independence**.

In accordance with the idea of encapsulation, objects are sometimes described, a little loosely, as having a **private memory** and a **public interface:**

- The private memory consists of **instance variables** (also known as *members* or *attributes*), whose values represent the internal state of the object. In a "pure" OO system, instance variables are completely private and hidden from users, but—as indicated above—they are of course visible to the code that implements the methods. However, we should add that many OO systems are *not* "pure" in this sense but do expose certain of their instance variables to users, a point we will return to later.
- The public interface consists of interface definitions—i.e., definitions of the inputs and outputs—for the methods that apply to this object. The code that implements those methods, like the instance variables, is hidden from the user. *Note:* It would be more accurate to say that the public interface is part of the **class-defining object** for the object in question (i.e., the object that defines the class of which the object in question is an instance), rather than part of the object itself. The code that implements those methods is kept with the class-defining object, not with the object itself.

Incidentally, note that if it is really true that predefined methods are the only way to manipulate objects, then *ad hoc* query is impossible!—unless objects are designed in accordance with a certain very specific discipline, another topic we will come back to later (in Chapter 24).

Methods are invoked by means of **messages**. A message is essentially just a function call, possibly asynchronous, in which one argument (the *receiver* or *target*) is distinguished and given special syntactic treatment. For instance, the following might be a message to department D, asking it to hire employee E:

```
D HIRE_EMP ( E )
```

(hypothetical syntax; see Section 22.6 for an explanation of the arguments D and E). The *receiver* or *target* in this example is the object denoted by D. A conventional programming language analog of this message might look like this:

```
HIRE_EMP ( D, E )
```

For convenience, an OO system will typically come equipped with certain builtin classes and methods. In particular, the system will almost certainly provide such classes as NUMERIC (with methods "=", "<", "+", "-", etc.), CHAR (with methods "=", "<", "||" (string concatenation), SUBSTRING, etc.), and so forth.

Instance Variables

We now take a slightly closer look at the concept of instance variables. The fact is, there is a certain amount of confusion in this area. As stated previously, instance variables would be completely hidden from the user in a pure OO system, but many systems are not pure. As a consequence, it is necessary to distinguish between *public* and *private* instance variables.

For example, suppose we have an object class of LINESEGs (line segments). Then the user might be able to think of each individual LINESEG as having a START point and an END point, and be able, e.g., to ask what the START point is for a given LINESEG. Internally, however, LINESEGs might be represented by MIDPOINT, LENGTH, and SLOPE, and these would be the variables seen by the code that implements the LINESEG methods (whatever those might be). In this example, then, START and END would be **public** instance variables and MIDPOINT, LENGTH, and SLOPE would be **private** instance variables.

It should be pointed out that—the foregoing notwithstanding—the concept of public instance variables is logically unnecessary. That is, even if the user is allowed to write an expression such as (say) *ls*.START to refer to the START point of LINESEG *ls*, that expression can still be regarded as shorthand for a function call that invokes a *method* called START that returns the START point for the LINESEG *ls*. In accordance with common practice, however, we will normally assume (until further notice) that objects typically do have one or more public instance variables.

It is worth pointing out too that just because certain "instance variable" arguments happen to be required in order to create objects of a particular class, it does not follow that those instance variables are available for arbitrary purposes. Suppose, for example, that creating a LINESEG requires START and END values. It does not follow that we can ask for, e.g., all LINESEGs with a given START value; whether this is possible will depend on whether a suitable method has been defined.

Finally, note that some systems support a variation on private instance variables called **protected** instance variables. If objects of class *C* have a protected instance variable *P*, then *P* is visible to the methods defined for class *C* (of course) *and* to the methods defined for any *subclass* (at any level) of class *C*. See Section 22.7 for a discussion of subclasses.

Object Identity

Every object has a unique **identity** called its "object ID" or OID. Primitive ("immutable") objects like the integer 3 are *self-identifying*, i.e., they are their own OIDs; other ("mutable") objects have (conceptual) *addresses* as their OIDs, and these addresses can be used elsewhere in the database as (conceptual) *pointers* to refer to the objects in

question. One implication—and one point of difference *vis-à-vis* the relational model—is that objects do not necessarily have to have any user-defined candidate keys. Note, however, that OIDs are not directly visible to the user. See Section 22.5 and Chapters 23–25 for further discussion.

We remark that it is sometimes claimed that it is an advantage of OO systems that two distinct objects can be identical in all user-visible respects—i.e., be duplicates of one another—and yet be distinguished via their OIDs. To this writer, however, this claim seems specious. For how can the *user* distinguish between two such objects, externally? (See references [4.5] and [4.11] for further discussion of this issue in a specifically relational context.)

22.4 Objects and Object Classes Revisited

Let us now consider a detailed example. Suppose we wish to create two object classes, DEPT (departments) and EMP (employees). Suppose also that the user-defined classes MONEY and JOB have already been created and the class CHAR is builtin. Then the necessary class creation operations for DEPT and EMP might look something like the following (hypothetical syntax):

```
CREATE OBJECT CLASS DEPT
       PUBLIC  ( DEPT#       CHAR,
                 DNAME       CHAR,
                 BUDGET      MONEY,
                 MGR         REF ( EMP ),
                 EMPS        REF ( SET ( REF ( EMP ) ) ) ) ...
       METHODS ( HIRE_EMP ( REF ( EMP ) ) ... code ... ,
                 FIRE_EMP ( REF ( EMP ) ) ... code ... , ... ) ... ;

CREATE OBJECT CLASS EMP
       PUBLIC  ( EMP#        CHAR,
                 ENAME       CHAR,
                 SALARY      MONEY,
                 POSITION    REF ( JOB ) ) ...
       METHODS ( ... ) ... ;
```

Points arising:

1. We have chosen to model departments and employees by means of a **containment hierarchy,** in which EMP objects are conceptually contained within DEPT objects. Thus, objects of class DEPT include a public instance variable called MGR, representing the given department's manager, and another one called EMPS representing the given department's employees. More precisely, objects of class DEPT include a public instance variable called MGR whose value is a reference ("REF") to an employee, and another one called EMPS whose value is a reference to a set of references to employees. ("Reference to" here really means *OID of*—see further discussion below.) We will elaborate on the containment hierarchy notion in a few moments.

2. For the sake of the example, we have chosen *not* to include a "foreign key" instance variable within objects of class EMP that refers to objects of class DEPT. This decision is consistent with our decision to model departments-and-employees by means of a containment hierarchy. However, it does mean that there is no direct way to get from a given EMP object to the corresponding DEPT object. See Chapter 24 (Section 24.3, subsection entitled "Relationships") for further discussion.
3. Note that each CREATE OBJECT CLASS operation includes definitions (coding details omitted) of the **methods** that apply to objects of that class.

Fig. 22.4 shows some sample object instances corresponding to the DEPT and EMP classes as just defined. Let us consider the EMP object at the top of that figure (with OID *eee*), which contains:

- The primitive value object "E001" (a character string) in the public instance variable EMP#
- The primitive value object "Smith" (another character string) in the public instance variable ENAME
- The user-defined object $50,000 (an object of class MONEY) in the public instance variable SALARY
- **The OID of** a user-defined object of class JOB in the public instance variable POSITION

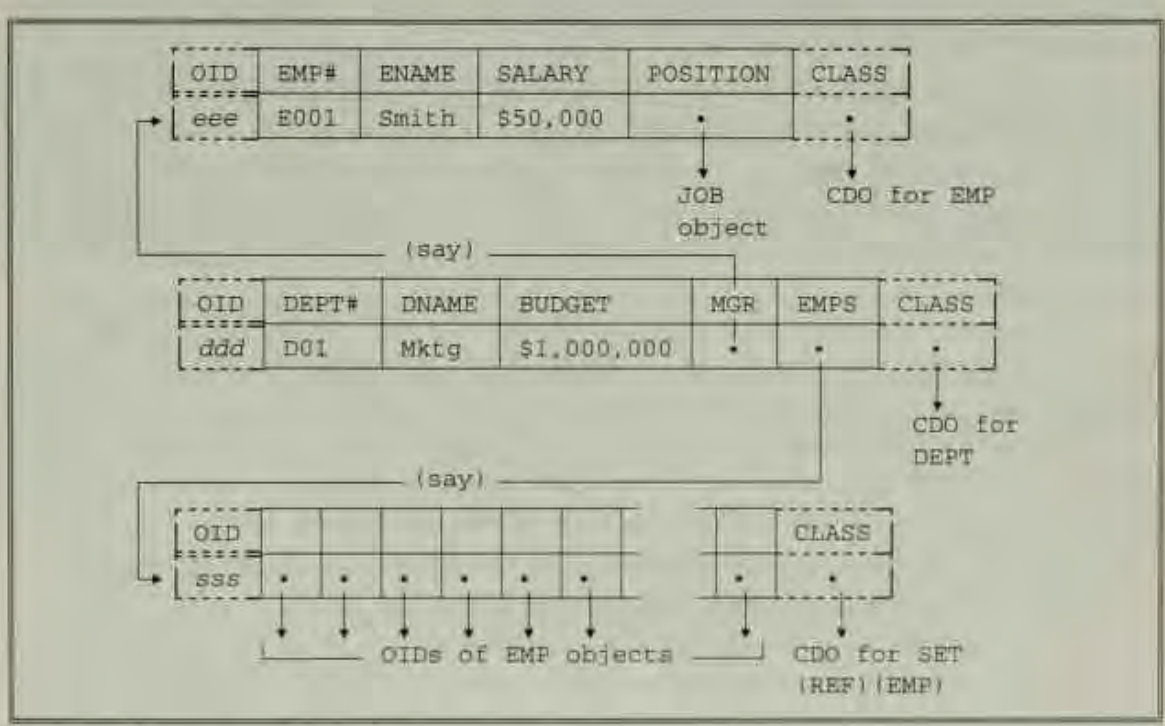

FIG. 22.4 Sample DEPT and EMP instances

It also contains at least two private instance variables, one that contains the OID *eee* of the EMP object itself, and one that contains the OID of the **class-defining object** (CDO) for EMPs (so that the implementation can find the code for the methods that apply to this object). *Note:* These two OIDs might or might not be physically stored with the object. For instance, the value *eee* need not necessarily be stored as part of the relevant EMP object; it is necessary only that the implementation have some way of locating that EMP object given that value *eee* (i.e., some way of mapping that value *eee* to that EMP object's physical address). But conceptually the user can always think of the OID as being part of the object, as shown.

Now let us turn to the DEPT object in the center of the figure (with OID *ddd*). That object contains:

- The primitive value object "D01" in the public instance variable DEPT#
- The primitive value object "Mktg" in the public instance variable DNAME
- The user-defined object $1,000,000 (an object of class MONEY) in the public instance variable BUDGET
- The OID *eee* of a user-defined object of class EMP in the public instance variable MGR (this is the OID of the object that represents the department manager)
- The OID *sss* of a user-defined object of class SET(REF(EMP)) in the public instance variable EMPS (see below)
- Two private instance variables containing in turn the OID *ddd* of the DEPT object itself and the OID of the corresponding class-defining object

And the object with OID *sss* consists of a set of OIDs of individual EMP objects (plus the usual private instance variables).

Now, Fig. 22.4 represents the sample instances "as they really are"; i.e., the figure illustrates the *data structure* component of "the OO model," and such figures must be clearly understood by users of that model. OO texts and presentations, however, typically do *not* show diagrams like that in Fig. 22.4; instead, they typically represent the situation as shown in Fig. 22.5 (which might be regarded as being at a higher level of abstraction).

The representation shown in Fig. 22.5 is certainly more consistent with the "containment hierarchy" interpretation. However, it obscures the important fact, already discussed above, that objects often contain, not other objects as such, but rather **OIDs** of other objects. For example, Fig. 22.5 clearly suggests that the DEPT object shown includes the EMP object for employee E001 *twice* (implying, among many other things, that employee E001 might be shown—inconsistently—as having two different salaries in its two different appearances). This sleight-of-hand is the source of much confusion, which is why we prefer pictures like that of Fig. 22.4.

As an aside, we note that real OO data definitions tend to increase the confusion, because they typically do *not* define instance variables as "REFs" (as in our hypothetical syntax above) but instead directly reflect the containment hierarchy interpretation. Thus, for example, instance variable EMPS in object class DEPT would typically be

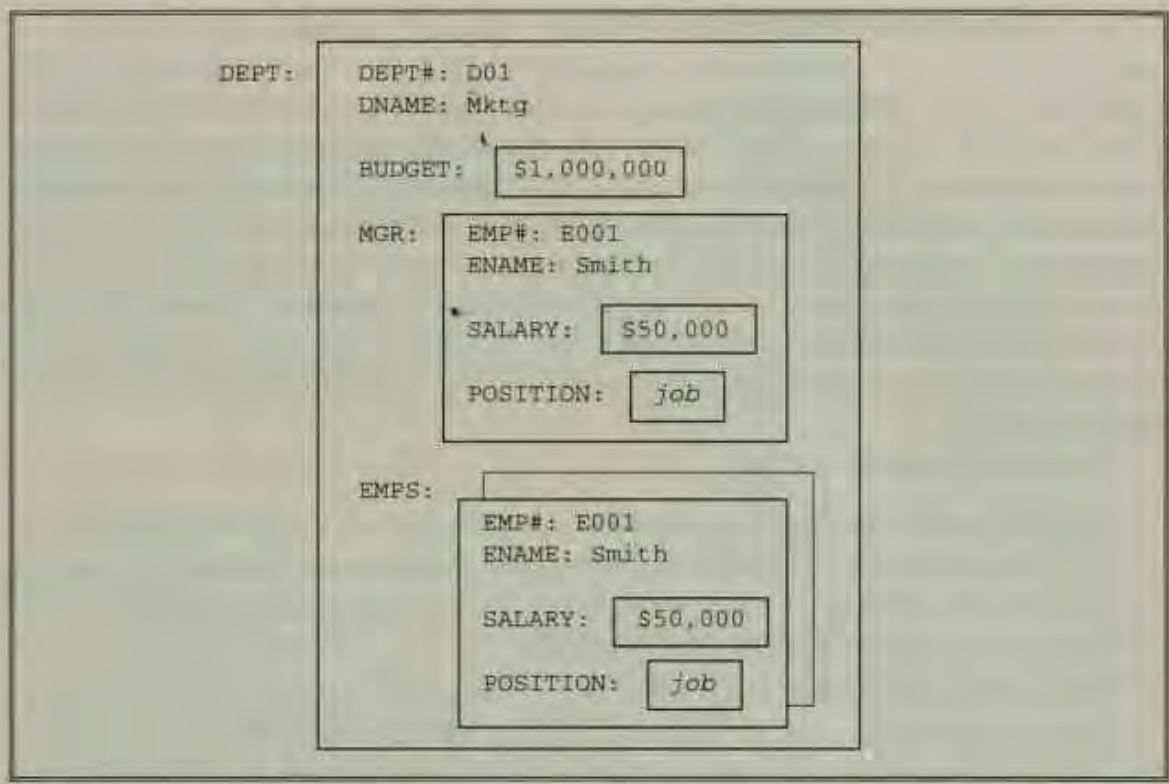

FIG. 22.5 Sample DEPT and EMP instances as a containment hierarchy

defined not as REF(SET(REF(EMP))) but just as SET(EMP). Although clumsy, we prefer our style of data definition, for clarity.

It is worth pointing out too that all of the old criticisms of hierarchies in general (as implemented in, e.g., IMS) apply to containment hierarchies in particular. Space does not permit detailed consideration of those criticisms here; suffice it to say that the overriding issue is **lack of symmetry**. In particular, hierarchies do not lend themselves well to the representation of many-to-many relationships. Consider suppliers and parts, for instance. Do the suppliers contain the parts, or *vice versa?* Or both? What about suppliers, parts, and projects?

Thus, it is our feeling, despite the common talk of "containment hierarchies," that objects are not really hierarchies at all. Instead, as Fig. 22.4 suggests, they are **tuples**—where the tuple components can be any of the following:

1. self-identifying values (e.g., integers)
2. simple nonshared "subobjects" (e.g., MONEY objects)
3. OIDs of other, possibly shared, "subobjects" (see Section 22.6)
4. sets, lists, arrays, . . . of 1., 2., 3., or 4.

(plus certain hidden OID, CDO OID, etc., components). Note carefully, however, that (in a pure system) such tuples are **encapsulated**.

22.5 Object Identity Revisited

Today's relational DBMSs typically rely on user-defined, user-controlled keys ("user keys" for short) for entity identification and referencing purposes. It is well known, however, that user keys suffer from a number of problems; reference [12.12] discusses such problems in some detail and argues that relational DBMSs should support *system*-defined keys ("surrogates") instead. And the argument in favor of OIDs in OO systems is very similar to the argument in favor of surrogates in relational systems. (Do not, however, make the mistake of equating the two: Surrogates are *values* and are visible to the user; OIDs are *addresses* or *pointers*—at least conceptually—and are hidden from the user.)

Points and questions arising:

1. First, it is very important to understand that OIDs do not avoid the need for user keys, as we will see in Chapter 23. To be more precise, user keys are still needed for interaction with the outside world, although inside the database all object cross-referencing can now be performed via OIDs.
2. What is the OID for a *derived* object?—e.g., the join of a given EMP and the corresponding DEPT, or the projection of a given DEPT over BUDGET and MGR? This is another important question that we will have to defer for now. See Chapter 25 (Section 25.4).
3. OIDs are the source of the oft-heard criticisms to the effect that OO systems look like "CODASYL warmed over"—where "CODASYL" refers generically to certain network (i.e., prerelational) database systems such as IDMS. Certainly OIDs tend to lead to a rather low-level, pointer-chasing style of programming (see Section 22.6 and Chapter 23) that is very reminiscent of the old CODASYL style. Also, the fact that OIDs are pointers and not values accounts for the claims, also sometimes heard, to the effect that CODASYL systems are closer to OO systems than relational systems are.

22.6 Classes, Instances, and Collections

OO systems make clear distinctions among the concepts of *class*, *instance*, and *collection*. First of all, an object **class** is (as previously explained) basically a data type, possibly builtin, possibly user-defined, of arbitrary internal complexity. (As an aside, however, we remark that—as mentioned in Section 22.3—some OO systems use both "type" and "class," in which case "type" means *data type* and "class" means *collection*. We will not adopt this usage in this book.)

Every class understands a **NEW** message, which causes a new **instance** of the class to be created (the method invoked by the NEW message is sometimes referred to as a *constructor function*). For example (hypothetical syntax):

```
E := EMP NEW ( 'E001', 'Smith', $50000, POS ) ;
```

Here POS is a program variable that contains the OID of some JOB object. The NEW method is invoked on the object class EMP; it creates a new instance of that class, initializes it to the specified values, and returns that new instance's OID. That OID is then assigned to the program variable E.

Next, because objects can contain pointers (OIDs) to other objects instead of objects *per se*, the very same object can be *shared* by many objects. In particular, the very same object can belong to multiple **collection** objects simultaneously. To continue with our example:

```
CREATE OBJECT CLASS EMP_COLL_CLASS
       PUBLIC ( EMP_COLL REF ( SET ( REF ( EMP ) ) ) ) ... ;

ALLEMPS  :=  EMP_COLL_CLASS NEW ( ) ;

ALLEMPS ADD ( E ) ;
```

Explanation:

- An object of class EMP_COLL_CLASS contains a single public instance variable, called EMP_COLL, whose value is a pointer (OID) to an object whose value is a set of pointers (OIDs) to individual EMP objects.
- ALLEMPS is a variable whose value is the OID of an object of class EMP_COLL_CLASS. After the assignment operation, it contains the OID of such an object whose value in turn is the OID of an *empty* set of OIDs of EMP objects.
- ADD is a method that is understood by objects of class EMP_COLL_CLASS. In the example, that method is applied to the object of that class whose OID is given in the variable ALLEMPS; its effect is to append the OID of the EMP object whose OID is given in the variable E to the (previously empty) set of OIDs whose OID is given in the EMP_COLL_CLASS object whose OID is given in the variable ALLEMPS.

After the foregoing sequence of operations, we can say, loosely, that the variable ALLEMPS denotes a collection of EMPs that currently contains just one EMP, namely employee E001. By the way, note the need to mention a user key value in this latter sentence!

Of course, we can have any number of distinct, and possibly overlapping, "sets of employees" at any given time:

```
PROGRAMMERS  :=  EMP_COLL_CLASS NEW ( ) ;
PROGRAMMERS ADD ( E ) ;
    .....
HIGHLY_PAID  :=  EMP_COLL_CLASS NEW ( ) ;
HIGHLY_PAID ADD ( E ) ;
```

and so on. Contrast the state of affairs in relational systems. For example, the SQL statement

```
CREATE TABLE EMP
     ( EMP# ... ,
       ENAME ... ,
       SALARY ... ,
       POSITION ... ) ... ;
```

creates a type *and* a collection simultaneously; the (composite) type corresponds to the table heading, and the (initially empty) collection corresponds to the table body. Likewise, the SQL statement

```
INSERT INTO EMP ( ... ) VALUES ( ... ) ;
```

creates an individual EMP row *and* appends it to the EMP collection simultaneously. In SQL, therefore:

1. There is no way for an individual EMP "object" to exist without being part of some "collection"—in fact, exactly one "collection" (but see below).
2. There is no direct way to create two distinct "collections" of the same "class" of EMP "objects" (but see below).
3. There is no direct way to share the same "object" across multiple "collections" of EMP "objects" (but see below).

At least, the foregoing are claims that are sometimes heard. In fact, however, they do not stand up to close scrutiny. To be specific, the relational foreign key mechanism can be used to achieve an equivalent effect in each case; for example, we could define two more base tables called PROGRAMMERS and HIGHLY_PAID, each of them consisting of just the employee numbers for the relevant employees. Alternatively, the relational view mechanism can be used to achieve a similar effect. For example, we could define PROGRAMMERS and HIGHLY_PAID as views of the EMP base table:

```
CREATE VIEW PROGRAMMERS
    AS SELECT EMP#, ENAME, SALARY, POSITION
       FROM   EMP
       WHERE  POSITION = 'Programmer' ;

CREATE VIEW HIGHLY_PAID
    AS SELECT EMP#, ENAME, SALARY, POSITION
       FROM   EMP
       WHERE  SALARY > some threshold, say $75000 ;
```

And now, of course, it is perfectly possible for the very same employee "object" to belong to two or more "collections" simultaneously. What is more, membership in those collections that happen to be views is handled automatically by the system, not manually by the programmer.

We close this discussion by mentioning an illuminating parallel between the *objects* of OO systems and the **explicit dynamic variables** of certain programming language systems (PL/I's BASED variables are a case in point). Like an OO object, an explicit dynamic variable can have any number of distinct instances, the storage for which is allocated by explicit programmer action. Furthermore, those instances, again

like individual object instances, *have no name*, and thus can be referenced only through pointers. In PL/I, for example, we might write:

```
DCL XYZ ... BASED ;          /* XYZ is a BASED variable */
DCL P POINTER ;              /* P is a pointer variable */

ALLOCATE XYZ SET ( P ) ;     /* create a new XYZ instance */
                             /* and set P to point to it  */

P -> XYZ = 3 ;               /* assign the value 3 to the XYZ */
                             /* instance pointed to by P      */
```

(and so on). This PL/I code bears a striking resemblance to the OO code discussed earlier; in particular, the declaration of the BASED variable is akin to the creation of an object class, and the ALLOCATE operation is akin to the creation of a NEW instance of that class. We can thus see that the reason OIDs are necessary in "the OO model" is precisely because, in general, the objects they identify do not possess any other unique name—just like BASED variable instances in PL/I.

22.7 Class Hierarchies

No treatment of basic OO concepts would be complete without some discussion of **class hierarchies** (not to be confused with containment hierarchies). However, the OO "class hierarchy" concept is essentially the same as the type hierarchy concept discussed in Chapter 19 (Section 19.4); we therefore content ourselves here with a few brief definitions (paraphrased from Chapter 19, for the most part) and a couple of relevant observations.

First, object class *Y* is said to be a **subclass** of object class *X*—equivalently, object class *X* is said to be a **superclass** of object class *Y*—if and only if every object of class *Y* is necessarily an object of class *X* ("*Y* **ISA** *X*"). Objects of class *Y* then **inherit** the instance variables and methods that apply to class *X*. As a consequence, the user can always use a *Y* object wherever an *X* object is permitted (i.e., as an argument to various methods)—this is the principle of **substitutability**—and thereby take advantage of **code reusability**. *Note:* Inheriting instance variables is referred to as **structural** inheritance; inheriting methods is referred to as **behavioral** inheritance. The ability to apply the same method to different classes—or, rather, the ability to apply different methods with the same name to different classes (a class *X* method might need to be **redefined** for use with class *Y*)—is referred to as **polymorphism**.

Here is an example. Suppose RECTANGLE is a subclass of POLYGON, which in turn is a subclass of PLANE_FIGURE. The object class declarations might look as follows:

```
CREATE OBJECT CLASS PLANE_FIGURE
       PUBLIC ( AREA ... ) ...
       METHODS ( OVERLAP ... ) ... )
```

```
CREATE OBJECT CLASS POLYGON
       ISA ( PLANE_FIGURE )
       PUBLIC ( VERTICES ... ) ...
       METHOD ( COUNT_VERTICES ... ) ... ;

CREATE OBJECT CLASS RECTANGLE
       ISA ( POLYGON )
       PUBLIC ( DIAGONAL_LENGTH ... )
       METHODS ( TRANSPOSE ... ) ... ;
```

A sample RECTANGLE instance is shown in Fig. 22.6. Note in particular that it includes the instance variables of PLANE_FIGURE and POLYGON as well as the instance variables of RECTANGLE *per se*. Note too the implications for creating a "NEW" instance of RECTANGLE, and contrast the situation with Fig. 22.7, which shows a relational representation of the same data.

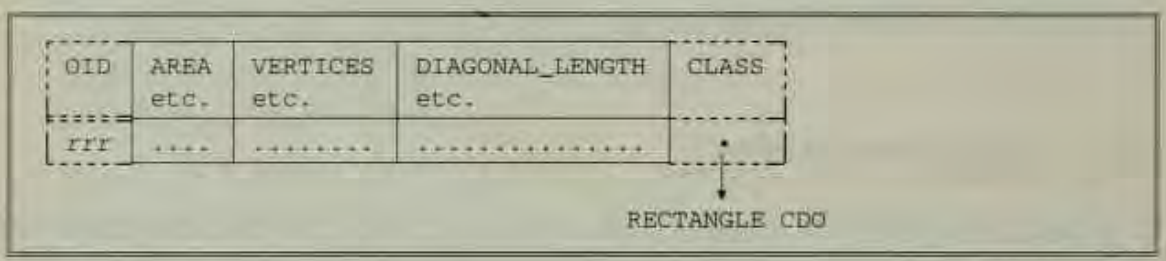

FIG. 22.6 Sample RECTANGLE instance

FIG. 22.7 Relational version of Fig. 22.6

The system will come equipped with some builtin class hierarchies. In OPAL, for example (see Chapter 23), every class is considered to be a subclass (at some level) of the builtin class OBJECT. Builtin subclasses of OBJECT include BOOLEAN, CHAR, INTEGER, COLLECTION (etc.); COLLECTION in turn has a subclass called BAG (a *bag* is the same as a set, except that it permits duplicates), and BAG has another called SET (etc., etc.).

Finally, we repeat another point from Chapter 19, *viz.*: Some systems support the notion of **multiple inheritance,** in which a given class can be a subclass of several superclasses simultaneously. In such a system, class "hierarchies" really become class **lattices,** in the sense that a given "child" class might have several distinct "parent" classes (but the "hierarchy" terminology is usually retained nonetheless).

22.8 Miscellaneous Remarks

In this section we make a few miscellaneous observations regarding objects in general.

1. A given object might acquire new types and lose existing types dynamically. For example, a given employee might become or cease to be a stockholder. The system must therefore support the ability for a given object to change its position within the class hierarchy—e.g., to "move up and down" a given branch, or (more complex) to be attached to or detached from an arbitrary node, at any time.
2. It is often suggested that OO technology simplifies database design and development (as well as database use) by providing high-level modeling constructs and supporting those constructs directly in the system. (By contrast, relational systems involve an extra level of indirection in mapping real-world objects into relations, attributes, foreign keys, etc.) But this claim raises the question: How exactly do we do OO database design? For instance, how do we decide whether to represent a given collection as a set or a bag or a list or an array? How do we decide whether to represent data explicitly (via instance variables) or procedurally (via methods)? And so on.
3. Finally, we remark that in an OO system, objects—that is, individual objects *per se*, not object classes—are:
   - the natural unit of security and authorization
   - the natural unit of recovery
   - the natural unit of concurrency

   (except where prohibited by the requirements of integrity constraints that span objects).

22.9 Summary

This brings us to the end of the first of our series of chapters on the OO approach. Following our discussion of the motivating RECTANGLES example, we explained:

- **Objects** (i.e., object **instances**), which correspond in traditional terms to *variables* (if "mutable") or *values* (if "immutable")
- **Object classes,** which correspond to *data types*
- **Methods** and **messages,** which correspond to *functions* and *calls* (possibly asynchronous calls), respectively

In addition, we pointed out that "mutable" objects correspond specifically to **explicit dynamic** variables, in programming parlance, which is why they have to be accessed through pointers (i.e., OIDs—see below).

Objects are **encapsulated** (i.e., their internal representation is not visible to users of the object; instead, users know only that the object can be manipulated via certain methods). Objects are sometimes described as having a **private memory** and a **public interface:** The private memory is the **instance variables,** the public interface is the definitions of the interfaces to the **methods**. In a "pure" OO system, only the methods—or, rather, the method *externals*—are visible to the user, but many systems are not pure. The methods are kept with the the **class-defining object** (CDO) for the object in question.

Every object has a unique **object ID** (OID). A given object's OID is used to refer to the object from other objects. Program variables also refer to objects by means of their OID. We stressed the point that an OID is a *pointer*, not a *value*, and left some rather important questions regarding OIDs unanswered. See Chapter 25 for further discussion.

We also described **containment hierarchies**—despite our feeling that the concept is misleading; in fact, it would be more accurate to regard objects not as hierarchies, but as (encapsulated) **tuples,** where the tuple components can be (a) self-identifying values, (b) simple nonshared "subobjects," (c) OIDs of other (possibly shared) subobjects, or (d) various combinations of the foregoing.

We then elaborated on the concepts of **class, instance,** and **collection.** OO systems make clear distinctions among these concepts; we gave some simple coding examples to illustrate the point, and compared and contrasted the situation with relational systems.

We then briefly touched on the OO **class hierarchy** concept, without however going into very much detail (because the concept has already been described, in slightly different terms, in Chapter 19). Finally, we offered a few further miscellaneous comments regarding objects in general.

Exercises

22.1 What is an object?

22.2 What does the term *encapsulated* mean?

22.3 Instance variables can be *public*, *private*, or *protected*. Explain.

22.4 What is the difference between *class* and *type*?

22.5 Define the concept *object ID*. What are the advantages of OIDs? What are the disadvantages?

22.6 How might OIDs be implemented?

22.7 What is a class-defining object?

22.8 What do you understand by the term "constructor function"? What analogs (if any) of such functions exist in traditional programming languages?

22.9 Explain and justify the *containment hierarchy* concept.

22.10 Explain the idea and conceptual implementation of "shared subobjects."

22.11 Give an example of multiple inheritance.

22.12 In Section 22.2 we gave two SQL formulations—a long form and a short form—of the query "Find all rectangles that overlap the unit square." Prove that those two formulations are equivalent.

References and Bibliography

References [22.1–22.3], [22.6], and [22.11] are textbooks on OO topics and related matters. References [22.9] and [22.16] are collections of research papers. References [22.7–22.8], [22.10], and [22.12] are tutorials.

22.1 Grady Booch. *Object Oriented Design, with Applications*. Redwood City, Calif.: Benjamin/Cummings (1991).

22.2 R. G. G. Cattell. *Object Data Management*. Reading, Mass.: Addison-Wesley (1991).

The first book-length tutorial on the application of OO technology to databases specifically. The following edited extract suggests that the field is still a long way from any kind of consensus: "Programming languages may need new syntax . . . swizzling [see Chapter 24, Section 24.2], replication, and new access methods also need further study . . . new end-user and application development tools [are] required . . . more powerful query-language features [must be] developed . . . new research in concurrency control is needed . . . timestamps and object-based concurrency semantics need more exploration . . . performance models are needed . . . new work in knowledge management needs to be integrated with object and data management capabilities . . . this [will lead to] a complex optimization problem [and] few researchers have [the necessary] expertise . . . federated [OO] databases require more study."

22.3 Brad J. Cox. *Object Oriented Programming: An Evolutionary Approach*. Reading, Mass.: Addison-Wesley (1986).

A tutorial text on object-oriented ideas in the programming world, with the emphasis on the use of OO techniques for software engineering.

22.4 O. J. Dahl, B. Myhrhaug, and K. Nygaard. *The SIMULA 67 Common Base Language*. Pub. S-22. Norwegian Computing Center, Oslo, Norway (1970).

SIMULA 67 was a language designed expressly for writing simulation applications. OO programming languages grew out of such languages; in fact, SIMULA 67 was really the first OO language.

22.5 C. J. Date. "An Optimization Problem." In C. J. Date and Hugh Darwen, *Relational Database Writings 1989–1991*. Reading, Mass.: Addison-Wesley (1992).

22.6 Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Reading, Mass.: Addison-Wesley (1983).

The definitive account of the pioneering efforts at the Xerox Palo Alto Research Center to design and build the Smalltalk-80 system. The first part of the book (out of four parts) is a detailed description of the Smalltalk-80 programming language, on which the OPAL language of GemStone is based (see Chapter 23).

22.7 Nathan Goodman. "Object Oriented Database Systems." *InfoDB 4*, No. 3 (Fall 1989).

See the annotation to this paper in Chapter 25 (reference [25.5]).

22.8 Nathan Goodman. "The Object Database Debate," "The Object Data Model," and "The Object Data Model in Action." *InfoDB 5*, No. 4 (Winter 1990–91); *6*, No. 1 (Spring/Summer 1991); and *6*, No. 2 (Fall 1991).

22.9 Won Kim and Frederick H. Lochovsky (eds.). *Object-Oriented Concepts, Databases, and Applications*. Reading, Mass.: Addison-Wesley (1989).

22.10 Roger King. "My Cat Is Object-Oriented." In reference [22.9].

22.11 Kamran Parsaye, Mark Chignell, Setrag Koshafian, and Harry Wong. *Intelligent Databases*. New York, N.Y.: John Wiley & Sons (1989).

22.12 Jacob Stein and David Maier. "Concepts in Object-Oriented Data Management." *Database Programming & Design 1*, No. 4 (April 1988).

A good tutorial on OO concepts by two of the GemStone designers.

22.13 Michael Stonebraker. *Introduction to* "New Data Models"; and *Introduction to* "Extendibility." Both in M. Stonebraker (ed.): *Readings in Database Systems*. San Mateo, Calif.: Morgan Kaufmann (1988).

These two introductions include Stonebraker's translation of OO concepts to traditional terms and his own brief analysis of what OO technology is trying to do and where it is likely to lead in the future.

22.14 D. C. Tsichritzis and O. M. Nierstrasz. "Directions in OO Research." In reference [22.9].

This paper lends further weight to the contention of the present writer that OO database technology is a long way from consensus: "There are disagreements on basic definitions; e.g., what is an object? . . . There is no reason to worry: Loose definitions are inevitable and sometimes welcome during a dynamic period of scientific discovery. They should and will become more rigorous during a period of consolidation that will inevitably follow." But OO concepts have been around for well over 25 years! (see reference [22.4]).

22.15 Gottfried Vossen. *Data Models, Database Languages, and Database Management Systems*. Reading, Mass.: Addison-Wesley (1991).

22.16 Stanley B. Zdonik and David Maier (eds.). *Readings in Object-Oriented Database Systems*. San Mateo, Calif.: Morgan Kaufmann (1990).

Answers to Selected Exercises

22.1 Part of the problem in discussing OO technology is that some of the most fundamental OO concepts lack a clear, universally accepted definition. Here, for example, are some "definitions" from the OO literature of the term **object:**

- "Objects are reusable modules of code that store data, information about relationships between data and applications, and processes that control data and relationships" (from a commercial product announcement).
- "An object is a chunk of private memory with a public interface" [22.12].
- "An object is an abstract machine that defines a protocol through which users of the object may interact" (introduction to reference [22.16]).
- "An object is a software structure that contains data and programs" [22.7].
- " . . . everything is an object . . . an object has a private memory and a public interface" [22.15].
- ". . the object-oriented model is based on a collection of objects. An object contains

values stored in *instance variables* within the object . . . these values are themselves objects . . . An object also contains bodies of code that operate on the object . . . called methods" [3.7].

Our own understanding of what an object is is explained in Section 22.3.

22.5 Some of the advantages of OIDs are as follows:

- They are not "intelligent." See reference [12.8] for an explanation of why this state of affairs is desirable.
- All objects in the database are identified in the same uniform way (contrast the situation with relational databases).
- They never change so long as the object they identify remains in existence.
- There is no need to repeat user keys in referencing objects. There is thus no need for an "ON UPDATE" foreign key rule.
- They are noncomposite. See references [5.5–5.7] for an explanation of why this state of affairs is desirable.

Some of the *dis*advantages—the fact that they do not avoid the need for user keys, the fact that they lead to a low-level "pointer-chasing" style of programming, and the fact that they apply to "base" objects only—were discussed briefly in Section 22.5.

22.6 Possible implementation techniques include:

- Physical disk addresses (fast but poor data independence).
- Logical disk addresses (i.e., page and offset addresses; fairly fast, better data independence).
- Artificial IDs (e.g., timestamps, sequence numbers; need mapping to actual addresses).

22.12 See reference [22.5].

23 A Cradle-to-Grave Example

23.1 Introduction

In the previous chapter we introduced the basic concepts of the OO approach. Now we show how those concepts fit together by presenting a "cradle-to-grave" example—i.e., we show how an OO database is defined, how it is populated, and how retrieval and update operations can be performed against that database. Our example is based on one of the best known OO systems, namely **GemStone** (available from Servio Corporation [23.2]) and its data language **OPAL**. *Note:* OPAL in turn was heavily influenced by Smalltalk-80 (see references [22.6] and [23.1]).

We introduce the example below. Section 23.2 then discusses data definition operations and Sections 23.3–23.5 discuss data manipulation operations. Section 23.6 presents a summary and a few additional comments.

The Example

The example is based on a simplified version of the education database from Exercise 5.3 in Chapter 5. To recap:

- An education database contains information about an inhouse company education training scheme. For each training course, the database contains details of all offerings of that course; for each offering it contains details of all student enrollments and all teachers for that offering. The database also contains information about employees.

A relational version of this database looks something like this:

```
EMPLOYEE ( EMP#, ENAME, JOB )
         PRIMARY KEY ( EMP# )

COURSE ( COURSE#, TITLE )
       PRIMARY KEY ( COURSE# )

OFFERING ( COURSE#, OFF#, ODATE, LOCATION )
         PRIMARY KEY ( COURSE#, OFF# )
         FOREIGN KEY ( COURSE# ) REFERENCES COURSE
```

```
ENROLLMENT ( COURSE#, OFF#, EMP#, GRADE )
           PRIMARY KEY ( COURSE#, OFF#, EMP# )
           FOREIGN KEY ( COURSE#, OFF# ) REFERENCES OFFERING
           FOREIGN KEY ( EMP# ) REFERENCES EMP

TEACHER ( COURSE#, OFF#, EMP# )
        PRIMARY KEY ( COURSE#, OFF#, EMP# )
        FOREIGN KEY ( COURSE#, OFF# ) REFERENCES OFFERING
        FOREIGN KEY ( EMP# ) REFERENCES EMP
```

Fig. 23.1 gives a referential diagram for this database. Refer to the Exercises and Answers in Chapter 5 if you require further explanation.

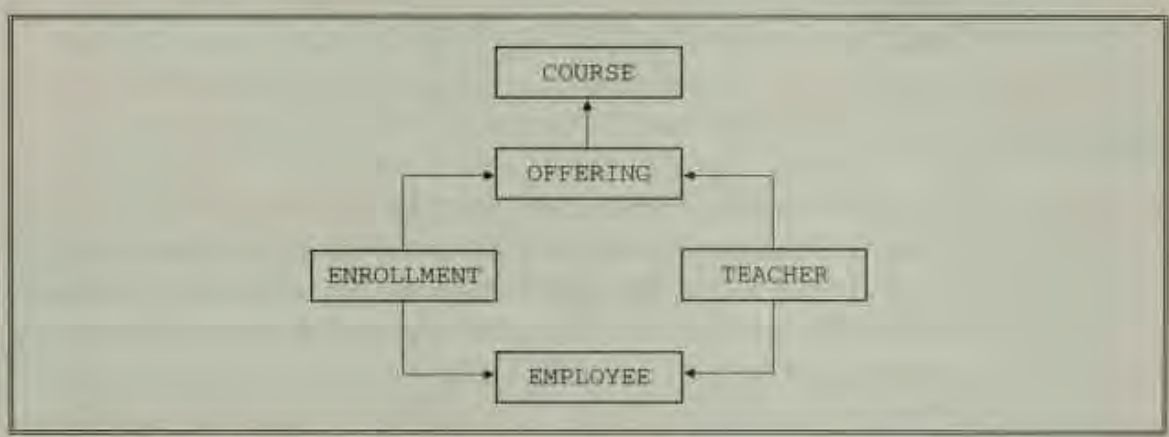

FIG. 23.1 Referential diagram for the education database

23.2 Data Definition

We now proceed to show an OPAL definition for the education database. Here first is the definition for an object class called EMPLOYEE (the lines are numbered for purposes of subsequent reference):

```
1 OBJECT SUBCLASS : 'EMPLOYEE'
2    INSTVARNAMES : # [ 'EMP#', 'ENAME', 'JOB' ]
3    CONSTRAINTS  : # [ # [ # EMP#, STRING ] ,
4                       [ # ENAME, STRING ] ,
5                       [ # JOB, STRING ] ] .
```

Explanation:

- Line 1 defines an object class called EMPLOYEE, a subclass of the builtin class called OBJECT. (In OPAL terms, line 1 is sending a message to the object called OBJECT, asking it to execute the method called SUBCLASS; INSTVARNAMES and CONSTRAINTS specify parameters to that method. Defining a new class—like everything else—is thus done by sending a message to an object.)

- Line 2 states that objects of class EMPLOYEE have three public instance variables called EMP#, ENAME, and JOB, respectively.
- Line 3 constrains the instance variable EMP# to contain objects of class STRING. Lines 4 and 5 are analogous.

Note: Throughout this chapter we omit discussion of purely syntactic details (such as the ubiquitous "#" signs in the foregoing example) that are essentially irrelevant for our purposes.

Next the COURSE class:

```
1 OBJECT SUBCLASS : 'COURSE'
2    INSTVARNAMES : # [ 'COURSE#', 'TITLE', 'OFFERINGS' ]
3    CONSTRAINTS  : # [ # [ # COURSE#, STRING ] ,
4                       [ # TITLE, STRING ] ,
5                       [ # OFFERINGS, OSET ] ] .
```

Explanation:

- Line 5 here specifies that instance variable OFFERINGS will contain an object of class OSET—or, more accurately, the OID of an object of class OSET.* Informally, OFFERINGS will denote the set of all offerings for the course in question. In other words, we have chosen to model the course-offerings relationship by means of a **containment hierarchy,** in which offerings are conceptually contained within the corresponding course. See later for the definition of the class OSET.

Now the OFFERING class:

```
1 OBJECT SUBCLASS : 'OFFERING'
2    INSTVARNAMES : # [ 'OFF#', 'ODATE', 'LOCATION',
3                                'ENROLLMENTS', 'TEACHERS' ]
4    CONSTRAINTS  : # [ # [ # OFF#, STRING ] ,
5                       [ # ODATE, DATETIME ] ,
6                       [ # LOCATION, STRING ] ,
7                       [ # ENROLLMENTS, NSET ] ,
8                       [ # TEACHERS, TSET ] ] .
```

Explanation:

- Line 7 specifies that instance variable ENROLLMENTS will contain (the OID of) an object of class NSET; informally, NSET will denote the set of all enrollments for the offering in question. Likewise, TSET will denote the set of all teachers for the offering in question. Again, therefore, we are adopting a "containment hierarchy" representation. See later for the NSET and TSET definitions.
- Note that we have chosen not to include a "foreign key" instance variable within objects of class OFFERING referring back to objects of class COURSE. This decision is consistent with our decision to use a containment hierarchy. However, it

* The hypothetical syntax introduced in Chapter 22 would have made this point more explicit, defining the instance variable OFFERINGS to be of type REF(SET(REF(OFFERING))).

does mean that there is no direct way to get from a given OFFERING instance to the corresponding COURSE instance. We will return to this issue in the next chapter.

Now the ENROLLMENT class:

```
1 OBJECT SUBCLASS : 'ENROLLMENT'
2    INSTVARNAMES : # [ 'EMP', 'GRADE' ]
3    CONSTRAINTS  : # [ # [ # EMP, EMPLOYEE ] ,
4                     # [ # GRADE, STRING ] ] .
```

Explanation:

- Again, objects of class ENROLLMENT do not include a "foreign key" instance variable referencing objects of class OFFERING. Note, however, that objects of class ENROLLMENT do include a *kind* of foreign key: Instance variable EMP (line 3) contains the OID of an object of class EMPLOYEE, representing the individual employee to whom this enrollment pertains. *Note:* The "containment hierarchy" interpretation would regard that EMPLOYEE object as actually being contained within the ENROLLMENT object, of course. But note the asymmetry: Enrollments are a many-to-many relationship, but the two participants in that relationship (employees and offerings) are treated quite differently.

Now we turn to teachers. For the sake of the example, we depart slightly from the original relational version of the database and treat teachers as a **subclass** of employees:

```
1 EMPLOYEE SUBCLASS : 'TEACHER'
2    INSTVARNAMES   : # [ 'COURSES' ]
3    CONSTRAINTS    : # [ # [ # COURSES, CSET ] ] .
```

Explanation:

- Line 1 defines an object class called TEACHER, a subclass of the previously defined class EMPLOYEE (in other words, TEACHER "ISA" EMPLOYEE). Thus, each individual TEACHER object has instance variables EMP#, ENAME, and JOB (all inherited from EMPLOYEE), plus COURSES, which will contain the OID of an object of class CSET. This CSET object will denote the set of all courses this teacher can teach.

Now, we saw in Chapter 22 that OO systems make a clear distinction between *classes* and **collections**. Indeed, the foregoing class definitions assumed the existence of several object collections—collections of offerings, collections of enrollments, collections of teachers, and so on—but no such collections have yet been defined. Let us therefore define a "collection" object class for each of the five classes defined above; we will name them ESET, CSET, OSET, NSET, and TSET, respectively. An object of class CSET, for example, will consist of a set of OIDs for individual objects of class COURSE. Here then are the definitions:

```
1 SET SUBCLASS    : 'ESET'
2     CONSTRAINTS : EMPLOYEE .
```

Explanation:

- Line 1 defines an object class called ESET, a subclass of the builtin class called SET.
- Line 2 constrains objects of class ESET to be sets of (OIDs of) objects of class EMPLOYEE. In general, there could be any number of objects of class ESET, but we will create just one (see Section 23.3), which will contain the set of OIDs of *all* EMPLOYEE objects (referred to loosely in what follows as "the set of all employees"). Informally, that ESET object will be the OO analog of the EMPLOYEE base relation in the relational version of the database.
- Observe that objects of class ESET have no public instance variables.

The CSET, OSET, NSET, and TSET definitions are analogous (see below). In each of these cases, however, we will definitely have to create several objects of the relevant class, not just one. For example, there will be as many OSET objects as there are individual COURSE objects. See Section 23.3 for the details of actually creating these several "set" objects.

```
SET SUBCLASS     : 'CSET'
    CONSTRAINTS : COURSE .

SET SUBCLASS     : 'OSET'
    CONSTRAINTS : OFFERING .

SET SUBCLASS     : 'NSET'
    CONSTRAINTS : ENROLLMENT .

SET SUBCLASS     : 'TSET'
    CONSTRAINTS : TEACHER .
```

To conclude this section, Fig. 23.2 (opposite) gathers together all of the foregoing definitions (and reorders them) for ease of subsequent reference. That figure will serve as a basis for the manipulative examples to be presented in the rest of this chapter.

23.3 Populating the Database

Now we consider what is involved in populating this database. We consider each of the five basic object types (employees, courses, offerings, enrollments, teachers) in turn.

Employees

Recall that we intend to collect together the OIDs of all EMPLOYEE objects in an ESET object, so first we need to create that ESET object:

```
OID_OF_SET_OF_ALL_EMPS  :=  ESET NEW .
```

Explanation:

- The expression on the right-hand side of this assignment returns the OID of a new, empty instance of class ESET (i.e., an empty set of EMPLOYEE OIDs); the OID

```
SET SUBCLASS     : 'ESET'
    CONSTRAINTS  : EMPLOYEE .

SET SUBCLASS     : 'CSET'
    CONSTRAINTS  : COURSE .

SET SUBCLASS     : 'OSET'
    CONSTRAINTS  : OFFERING .

SET SUBCLASS     : 'NSET'
    CONSTRAINTS  : ENROLLMENT .

SET SUBCLASS     : 'TSET'
    CONSTRAINTS  : TEACHER .

OBJECT SUBCLASS : 'EMPLOYEE'
   INSTVARNAMES : # [ 'EMP#', 'ENAME', 'JOB' ]
   CONSTRAINTS  : # [ # [ # EMP#, STRING ] ,
                        [ # ENAME, STRING ] ,
                        [ # JOB, STRING ] ] .

OBJECT SUBCLASS : 'COURSE'
   INSTVARNAMES : # [ 'COURSE#', 'TITLE', 'OFFERINGS' ]
   CONSTRAINTS  : # [ # [ # COURSE#, STRING ] ,
                        [ # TITLE, STRING ] ,
                        [ # OFFERINGS, OSET ] ] .

OBJECT SUBCLASS : 'OFFERING'
   INSTVARNAMES : # [ 'OFF#', 'ODATE', 'LOCATION',
                              'ENROLLMENTS', 'TEACHERS' ]
   CONSTRAINTS  : # [ # [ # OFF#, STRING ] ,
                        [ # ODATE, DATETIME ] ,
                        [ # LOCATION, STRING ] ,
                        [ # ENROLLMENTS, NSET ] ,
                        [ # TEACHERS, TSET ] ] .

OBJECT SUBCLASS : 'ENROLLMENT'
   INSTVARNAMES : # [ 'EMP', 'GRADE' ]
   CONSTRAINTS  : # [ # [ # EMP, EMPLOYEE ] ,
                        [ # GRADE, STRING ] ] .

EMPLOYEE SUBCLASS : 'TEACHER'
   INSTVARNAMES   : # [ 'COURSES' ]
   CONSTRAINTS    : # [ # [ # COURSES, CSET ] ] .
```

FIG. 23.2 Data definition for the education database (OO version)

of that new instance is then assigned to the program variable OID_OF_SET_OF_ALL_EMPS. Informally, OID_OF_SET_OF_ALL_EMPS will be used to identify "the set of all employees."

Now, every time we create a new EMPLOYEE object we want the OID of that object to be inserted into the ESET object identified by the OID saved in the variable OID_OF_SET_OF_ALL_EMPS. We therefore define a **method** for creating such an EMPLOYEE object and inserting its OID into that ESET object:

```
 1 METHOD : ESET                          /* anonymous!      */
 2   ADD_EMP#   : EMP#_PARM               /* parameters      */
 3   ADD_ENAME  : ENAME_PARM
 4   ADD_JOB    : JOB_PARM
 5   | EMP_OID |                           /* local variable */
 6   EMP_OID := EMPLOYEE NEW .            /* new employee    */
 7   EMP_OID SET_EMP# : EMP#_PARM ,       /* initialize      */
 8           SET_ENAME : ENAME_PARM ,
 9           SET_JOB : JOB_PARM .
10   SELF ADD: EMP_OID .                  /* insert          */
11 %
```

Explanation:

- Line 1 defines the code that follows (up to the terminating percent sign in line 11) to be a method that applies to objects of class ESET. (In fact, of course, *exactly one* object of that class will exist in the system at run time.)
- Lines 2–4 define three parameters, with external names ADD_EMP#, ADD_ENAME, and ADD_JOB. These names will be used in messages that invoke the method. The corresponding internal names EMP#_PARM, ENAME_PARM, and JOB_PARM will be used in the code that implements the method.
- Line 5 defines EMP_OID to be a local variable, and line 6 then assigns to that variable the OID of a new, uninitialized EMPLOYEE instance.
- Lines 7–9 send a message to that new EMPLOYEE instance; the message specifies three methods (SET_EMP#, SET_ENAME, and SET_JOB) and passes one parameter to each of them (EMP#_PARM to SET_EMP#, ENAME_PARM to SET_ENAME, and JOB_PARM to SET_JOB). *Note:* We are assuming here that SET_EMP# is a method that applies to an EMPLOYEE object and has the effect of setting the EMP# instance variable within that object to the specified value, and similarly for SET_ENAME and SET_JOB. We will have a little more to say regarding these three methods in just a moment.
- Line 10 sends a message to SELF, which is a special variable that represents the object to which the method being defined is currently being applied (i.e., the current *receiver* or *target* object). The message causes the builtin method ADD to be applied to that object (ADD is a method that is understood by every "collection"-type class); the effect in the case at hand is to insert the OID of the object identified by EMP_OID into the object identified by SELF. *Note:* The reason the special

variable SELF is required is that the parameter corresponding to the receiver object is otherwise unnamed.

- Note that—as pointed out by the comment in line 1—the method being defined is likewise unnamed. In general, in fact, methods do not have names in OPAL, but instead are identified by their **signature** (i.e., the combination of the name of the class to which they apply and the external names of their parameters). This convention leads to awkward circumlocutions, as can be seen. Note too another slightly unfortunate implication—namely, that if two methods both apply to the same class and take the same parameters, those parameters must be given arbitrarily different external names in the two methods.

Now we have a method for inserting new EMPLOYEEs into the database, but we still have not actually inserted any. So let us do so:

```
OID_OF_SET_OF_ALL_EMPS ADD_EMP#   : 'E009',
                       ADD_ENAME  : 'Watt',
                       ADD_JOB    : 'Janitor' .
```

This statement (a) creates an EMPLOYEE object for employee E009 and (b) adds (the OID of) that EMPLOYEE object to the set of (OIDs of) all EMPLOYEE objects.

Note, incidentally, that the builtin NEW method must now never be used on class EMPLOYEE other than as part of the method we have just defined—for otherwise we might create some dangling EMPLOYEE objects, i.e., employees who are not represented in "the set of all employees." *Note:* We apologize for the burdensome repetition of cumbersome circumlocutions like "the set of all employees" and "the method we have just defined," but it is not easy to talk about things that have no name.

To conclude our discussion of the employees case, we must say something regarding the methods SET_EMP#, SET_ENAME, and SET_JOB. As explained in Chapter 22, a pure OO system will not expose instance variables to the user; instead, it will permit objects to be manipulated solely by means of methods (**encapsulation**). In accordance with this principle, we will adopt the convention throughout this chapter that instance variables are indeed hidden; if a class definition in fact defines a "public" instance variable *v*, we will treat that definition as shorthand for the definition of a pair of methods GET_*v* and SET_*v*, where GET_*v* returns the value of *v* to the caller and SET_*v* sets *v* to the value specified by the caller.

Courses

Employees really represent the simplest possible case, since they correspond to "regular entities" (to use the terminology of the E/R model), and moreover do not contain any other objects embedded within themselves. We now move on to consider the more complex case of *courses*, which—although still "regular entities"—do conceptually include certain other objects embedded within them. In outline, the steps we must go through are as follows:

1. Apply the NEW method to class CSET to create an initially empty "set of all courses" (actually COURSE OIDs).

2. Define a method for creating a new COURSE object and inserting its OID into "the set of all courses." That method will take a specified COURSE# and TITLE as parameters and will create a new COURSE object with the specified values. It will also apply the NEW method to class OSET to create an initially empty set of offerings (actually OFFERING OIDs), and will then place the OID of that empty set of offerings into the OFFERINGS position within the new COURSE object.
3. Invoke the method just defined for each individual course in turn.

Offerings

The "offerings" steps are as follows:

1. Define a method for creating a new OFFERING object. That method will take a specified OFF#, ODATE, and LOCATION as parameters and will create a new OFFERING object with those specified values. It will also:
   - Apply the NEW method to class NSET to create an initially empty set of enrollments (actually ENROLLMENT OIDs), and then place the OID of that empty set of enrollments into the ENROLLMENTS position within the new OFFERING object.
   - Apply the NEW method to class TSET to create an initially empty set of teachers (actually TEACHER OIDs), and then place the OID of that empty set of teachers into the TEACHERS position within the new OFFERING object.
2. The method will also take a COURSE# parameter, and will use that COURSE# value to:
   - Find the corresponding COURSE object for the new OFFERING object. See Section 23.4 for an explanation of how this might be done. *Note:* Of course, the method must reject the attempt to create a new offering if the corresponding course cannot be found. We omit further consideration of such exception cases from the rest of our discussions.
   - Hence, locate "the set of all offerings" for that COURSE object.
   - Hence, add the OID of the new OFFERING object to the appropriate "set of all offerings."

   Note carefully, therefore, that (as mentioned in Chapter 22) OIDs do *not* avoid the need for user keys such as COURSE#. Indeed, such keys are needed, not just to refer to objects in the outside world, but also to serve as the basis for lookups inside the database.
3. Finally, invoke the method just defined for each individual offering in turn.

Note, incidentally, that (in keeping with our containment hierarchy representation) we have chosen not to create a "set of *all* offerings." One implication of this omission is that any query that requires that set as its scope—e.g., "Find all offerings in New York"—will involve a certain amount of procedural workaround code.

Enrollments

The difference in kind between the enrollments and offerings cases is that ENROLLMENT objects include a instance variable of class EMPLOYEE—namely, the variable EMP, whose value is the OID of the relevant EMPLOYEE object. Hence the necessary sequence of steps is as follows:

1. Define a method for creating a new ENROLLMENT object. That method will take a specified COURSE#, OFF#, EMP# and GRADE as its parameters and will create a new ENROLLMENT object with the specified GRADE value. It will then:
   - Use the COURSE# and OFF# values to find the corresponding OFFERING object for the new ENROLLMENT object.
   - Hence, find "the set of all enrollments" for that OFFERING object.
   - Hence, add the OID of the new ENROLLMENT object to the appropriate "set of all enrollments."

   It will also:
   - Use the EMP# value to find the relevant EMPLOYEE object.
   - Hence, place the OID of that EMPLOYEE object in the EMP position within the new ENROLLMENT object.
2. Invoke the method just defined for each individual enrollment in turn.

Teachers

The difference in kind between the teachers and offerings cases is that TEACHER is a subclass of EMPLOYEE. Hence:

1. Define a method for creating a new TEACHER object. That method will take a specified COURSE#, OFF#, and EMP# as its parameters. It will then:
   - Use the EMP# value to find the relevant EMPLOYEE object.
   - Convert that EMPLOYEE object into a TEACHER object (since that employee is now additionally a teacher; refer back to Section 22.7 if you need to refresh your memory regarding the representation of superclass/subclass objects in an OO system).

   It will also:
   - Use the COURSE# and OFF# values to find the corresponding OFFERING object for the new TEACHER object.
   - Hence, find "the set of all teachers" for that OFFERING object.
   - Hence, add the OID of the new TEACHER object to the appropriate "set of all teachers."
2. In addition, the set of courses this teacher can teach must be specified somehow and the COURSES instance variable set appropriately in the new TEACHER object. We omit the details of this step here.
3. Invoke the method just defined for each individual teacher in turn.

Relational Analogs

For comparison purposes, we close this section with a reminder of what is involved in populating the *relational* version of the education database. Here, for example, is the SQL code to insert employee E009 into the database:

```
INSERT INTO EMPLOYEE ( EMP#, ENAME, JOB )
       VALUES ('E009', 'Watt', 'Janitor') ;
```

Some further examples:

```
INSERT INTO COURSE ( COURSE#, TITLE )
       VALUES ('C001', 'Database Technology') ;

INSERT INTO OFFERING ( COURSE#, OFF#, ODATE, LOCATION )
       VALUES ('C001', '004', DATE '94/01/18', 'CDG') ;

INSERT INTO ENROLLMENT ( COURSE#, OFF#, EMP#, GRADE )
       VALUES ('C001', '004', 'E009', '') ;
```

And so on. Note in particular that the information the (end) user must provide in order to insert, say, a new enrollment is *exactly the same* in both the relational and the OO case.

23.4 Retrieval Operations

Before getting into details of retrieval operations as such, we make the point (though it should already be obvious) that OPAL—like OO languages in general—is essentially record-at-a-time, not set-at-a-time. Hence, most problems will require some programmer to write a block of procedural code.

We consider a single example—the query "Find all New York offerings of course C001." We suppose for the sake of the example that we have a variable called OOSOAC whose value is the OID of "the set of all courses." Here then is the code:

```
1 | COURSE_C001 , C001_OFFS , C001_NY_OFFS |
2   COURSE_C001
3      := OOSOAC DETECT : [ :CX | 'C001' = CX GET_COURSE# ] .
4   C001_OFFS
5      := COURSE_C001 GET_OFFERINGS .
6   C001_NY_OFFS
7      := C001_OFFS SELECT : [ :OX | 'New York' = OX GET_LOCATION ] .
8 ^ C001_NY_OFFS .
```

Explanation:

- Line 1 declares three local variables—COURSE_C001, which will be used to hold the OID of course C001; C001_OFFS, which will be used to hold the OID of "the set of all offerings" for course C001; and C001_NY_OFFS, which will be used to hold the OID of the set of OIDs of the required offerings.
- Lines 2–3 send a message to the object denoted by the OOSOAC variable. The

message calls for the builtin **DETECT** method to be applied to that object. The DETECT argument is a block of code of the form

```
[ :x | p(x) ]
```

Here $p(x)$ is a conditional expression involving the variable x, and x is effectively a range variable that ranges over the members of the set to which the DETECT is applied (i.e., the set of COURSE objects, in the example). The result of the DETECT is the OID of the first object encountered in that set that makes $p(x)$ evaluate to *true*—i.e., it is the COURSE object for course C001, in the example. The OID of that COURSE object is then assigned to the variable COURSE_C001.

Note: It is also possible to specify an "escape" argument to DETECT to deal with the case in which $p(x)$ never evaluates to *true*. We omit the details here.

- Lines 4–5 assign the OID of "the set of all offerings" for course C001 to the variable C001_OFFS.
- Lines 6–7 are rather similar to lines 2–3: The operation of the builtin **SELECT** method is the same as that of DETECT, except that it returns the OID of the set of OIDs of *all* objects (instead of just the first) that make $p(x)$ evaluate to *true*. In the example, therefore, the effect is to assign the OID of the set of OIDs of New York offerings of course C001 to the variable C001_NY_OFFS.
- Finally, line 8 returns that OID to the caller (the symbol ^ is used to mean "Display" or "Return result").

Points arising from this example:

1. First, note that the block of code in SELECT and DETECT can involve (at its most complex) a set of simple scalar comparisons all ANDed together—i.e., it is a limited form of restriction condition.
2. The square brackets surrounding the block of code in SELECT and DETECT can be replaced by curly brackets (braces). If curly brackets are used, OPAL will attempt to use an index (if an appropriate index exists) in applying the method. If square brackets are used, it will not.
3. When we say that DETECT returns the OID of the "first" object encountered that makes $p(x)$ evaluate to *true*, we mean, of course, the first according to whatever sequence OPAL uses to search the set (sets have no intrinsic ordering of their own). In our example it makes no difference, because the "first" object that makes the condition *true* is in fact the *only* such.
4. The observant reader will have noticed that we have been using expressions such as "the DETECT method," whereas we pointed out previously that methods have no names in OPAL. Indeed, DETECT and SELECT are not method names (and phraseology such as "the DETECT method" is strictly incorrect). Rather, they are external parameter names for certain builtin (and unnamed) methods. For brevity and simplicity, however, we will continue to talk as if DETECT and SELECT (and other similar items) were indeed method names.
5. The observant reader might also have noticed that we have been using (many

times!) the expression "the NEW method." This usage is in fact *not* incorrect: Methods that take no arguments other than a receiver are an exception to the general OPAL rule that methods are unnamed.

Now, the foregoing simple example does not illustrate the point, but (because of the record-at-a-time level of the OPAL language) many retrieval problems will naturally require the execution of one or more *loops*. OPAL supports loops by means of a special **DO** method. A message that invokes DO is similar in form to one that invokes SELECT or DETECT:

```
object DO : [ :x | block-of-code ]
```

Here *x* ranges over the members of the set denoted by *object*, and *block-of-code* is executed for each such member in turn. Using GET_*v* and SET_*v* methods, that code can retrieve instance variables within the individual objects of the set (or update them, of course, but retrieval is the topic currently under discussion).

IFTRUE and **IFFALSE** methods are also available to support conditional code execution. We omit the details here.

23.5 Update Operations

The OO analog of INSERT has already been discussed in Section 23.3. The OO analogs of UPDATE and DELETE are discussed below.

Update

Updating operations are performed in essentially the same manner as retrieval operations, except that SET_*v* methods are used instead of GET_*v* methods.

Delete

The builtin method REMOVE is used to delete objects. More precisely, it is used to remove the OID of a specified object from a specified collection. When an object reaches a point when there are no remaining references to it—i.e., it cannot be accessed at all—then OPAL deletes it automatically by means of some kind of "garbage collection" process. Here is an example:

```
OID_OF_SET_OF_ALL_EMPS
   REMOVE : DETECT : [ :EX | 'E001' = EX GET_EMP# ] .
```

("remove employee E001 from the set of all employees").

But what if we want to enforce (say) a "DELETE CASCADES" rule?—for example, a rule to the effect that deleting an employee is to cascade and delete (e.g.) all enrollments for that employee as well? The answer, of course, is that we have to implement an appropriate method once again; in other words, we have to write some more procedural code.

Incidentally, it might be thought that the garbage collection approach to deletion

does at least implement a kind of "DELETE RESTRICTED" rule, inasmuch as an object is not actually deleted so long as any references to that object exist. However, such is not necessarily the case. For example, OFFERING objects do not include the OID of the corresponding COURSE object, and hence offerings do not "restrict" DELETEs on courses.

(In fact, containment hierarchies tacitly imply a kind of "DELETE CASCADES" rule, *unless* the user chooses either

- to include the parent OID in the child, or
- to include the child OID in some other object elsewhere in the database,

in which case the "containment hierarchy" interpretation no longer makes sense anyway. See the discussion of *inverse variables* in Chapter 24.)

Note finally that REMOVE can be used to emulate a relational DESTROY or SQL DROP operation—e.g., to destroy the ENROLLMENT class object. The details are left as an exercise (highly nontrivial) for the reader.

23.6 Summary

This brings us to the end of our "cradle-to-grave" example. Our example was based on **OPAL**, a dialect of **Smalltalk-80** supported by the **GemStone product**. We began by discussing **data definition** operations, showing how object classes are created in OPAL and pointing out that we typically need both (a) a "base" object class, such as EMPLOYEE, that allows us to create individual object instances, and (b) a "collection" object class, such as ESET, that allows us to gather together sets of such object instances into one or more collections. The reason both are needed, of course, is that the base object class is just a type declaration; thus, some collection object class is required to hold (pointers to) actual object instances.

We then considered in some detail what is involved in **populating** the database, showing how appropriate methods might be written. Those methods were fairly nontrivial, involving as they did a great deal of procedural adjustment of existing objects in order to maintain (OO analogs of) certain referential constraints. (*Note:* We will have more to say regarding referential constraints in the next chapter.) We stressed the point that *all* "insert" operations must be performed via those defined methods, not directly via the builtin NEW method. We also pointed out that user keys are still necessary; in particular, the information the end user has to provide on "insert" is exactly the same in the OO case as it would be in the relational case.

Next, we considered **retrieval** operations. Since OO systems are essentially record-at-a-time, retrieval operations (like everything else) tend to be fairly procedural in nature. However, we did give examples of the OPAL **DETECT** and **SELECT** methods, which are slightly less procedural than some.

Last, we considered **update** operations. "Update" *per se* is very similar to retrieval, except for the use of **SET_***y* methods instead of **GET_***y* methods. "Deletion" is done via the **REMOVE** method and a garbage collection process (in OPAL, that is; some

OO systems do support deletion in a more explicit manner). We briefly discussed the implications of this latter fact for foreign key rules such as RESTRICTED and CASCADES.

Miscellaneous Issues

We close this section (and this chapter) with a few miscellaneous remarks regarding OO database operations in general.

- First, consider the following two methods:

  1. Find the courses attended by employee *E*
  2. Find the employees attending course *C*

  These two methods are clearly inverses of each other, but they must be separately defined—i.e., two separate pieces of procedural code must be written. What is more, of course, matters get combinatorially worse as the number of unknowns increases. For instance, suppose we have courses, employees, and grades. Then each of the following queries will require a different method to be explicitly defined:

  1. Find the grade obtained by employee *E* on course *C*.
  2. Find the employees who obtained an "A" on course *C*.
  3. Find the courses on which employee *E* obtained an "A".
  4. Find the grades obtained by employee *E* on all courses.

  (etc., etc., etc.). Furthermore, it is not at all clear which object(s) these method(s) should be applied to; to find the grade obtained by employee *E* on course *C*, for example, do we send a message to *E* or to *C*? (See Chapter 24 for further discussion of this latter point.)

  The relational approach, of course, solves all of the foregoing problems very nicely. First, we have the ENROLLMENT relation, which represents all relevant information in a completely symmetric manner:

  ```
  ENROLLMENT ( COURSE#, OFF#, EMP#, GRADE )
  ```

  Each of the queries mentioned above can then be represented by a single relational request against this relation; there is no need to define lots of different methods, with different (in some cases, arbitrarily different) arguments. The details are left as an exercise to the reader. Relational systems thus provide **symmetric exploitation** [4.1]: The user can access any given relation with any combination of attribute values as "knowns" and retrieve the other attribute values as the corresponding "unknowns." OO systems, by contrast, do not provide symmetric exploitation.

- Note that we have said nothing so far in this chapter regarding relational operators such as **join**. What would it mean to join two objects? What class of object would the result be? What would its OID be?

  In fact, OO systems typically support **path expressions** instead of join. For example, we might write

  ```
  ENROLL . OFFERING . COURSE . COURSE#
  ```

  to obtain the course number of the course of the offering of the enrollment whose

OID is given in the variable ENROLL. Such an expression is the OO equivalent of a certain relational expression (involving two joins and a projection). But note that such "joins" are therefore limited (a) to predefined paths only, and (b) to many-to-one joins only. Note too that such "joins" do *not* yield new objects, i.e., they simply navigate through the database to locate existing objects.

- We must not conclude our discussion of OO database operations without considering the question of how much of what we have seen is intrinsic to OO systems in general and how much is due merely to quirks of OPAL specifically. It seems to this writer that—in principle, at least—certain of the complexities we have been examining could have been avoided; for example, the fact that receiver parameters are unnamed and the fact that methods are unnamed both seem to lead to unnecessary complications. But other aspects of "the OO model," such as the heavy emphasis on pointers and the record-at-a-time nature of programming and the fact that objects themselves are unnamed, seem to be more inherent, and some of the complexity we have seen is due to such matters also.

Exercises

23.1 Design an OO version of the suppliers-and-parts database. *Note:* This design will be used as a basis for Exercises 23.2–23.4 below.

23.2 Write a suitable set of OPAL data definition statements for your OO version of the suppliers-and-parts database.

23.3 Sketch the details of the necessary "database populating" methods for your OO version of the suppliers-and-parts database.

23.4 Write OPAL code for the following queries on your OO version of the suppliers-and-parts database:

(a) Get all London suppliers.

(b) Get all red parts.

23.5 Consider the education database once again. Show what is involved in (a) deleting an enrollment, (b) deleting an employee, (c) deleting a course (assuming in every case that the OPAL-style garbage collection process applies). State any assumptions you make regarding such matters as "DELETE CASCADES," "DELETE RESTRICTED," etc.

23.6 Suppose the suppliers-parts-projects database is to be represented by means of a single OO containment hierarchy. How many possible such hierarchies are there? Which one is best?

23.7 Consider a variation on the suppliers-parts-projects database in which, instead of recording that certain suppliers supply certain parts to certain projects, we wish to record only that (a) certain suppliers supply certain parts, (b) certain parts are supplied to certain projects, and (c) certain projects are supplied by certain suppliers. How many possible OO designs are there now (with or without containment hierarchies)?

References and Bibliography

23.1 George Copeland and David Maier. "Making Smalltalk a Database System." Proc. 1984 ACM SIGMOD International Conference on Management of Data, Boston, Mass. (June 1984). Republished in M. Stonebraker (ed.), *Readings in Database Systems*. San Mateo, Calif.: Morgan Kaufmann (1988).

Describes some of the enhancements and changes that were made to Smalltalk-80 [22.6] in order to create GemStone and OPAL.

23.2 Servio Corporation. *GemStone Reference Manual*. Beaverton, Ore. (1990).

Answers to Selected Exercises

23.5 We do not give a detailed answer to this question, but we do make one remark concerning its difficulty. First, let us agree to use the term "delete" as a shorthand to mean "make a candidate for physical deletion" (i.e., by erasing all references to the object in question). Then in order to delete an object *X*, we must first find all objects *Y* that include a reference to *X*; for each such object *Y*, we must then either delete that object *Y*, or at least erase the reference in that object *Y* to the object *X* (by setting that reference to the special value *nil*). And part of the problem is that it is not possible to tell from the data definition alone exactly which objects include a reference to *X*, nor even how many of them there are. Consider employees, for example, and the object class ESET. In principle, there could be any number of ESET instances, and any subset of those ESET instances could include a reference to some specific employee.

23.6 There are obviously six possible hierarchies:

```
S contains ( P contains ( J ) )
S contains ( J contains ( P ) )
P contains ( J contains ( S ) )
P contains ( S contains ( J ) )
J contains ( S contains ( P ) )
J contains ( P contains ( S ) )
```

"Which is best?" is unanswerable without additional information, but almost certainly *all* of them are bad.

23.7 There are at least twelve "obvious" containment hierarchy designs. Here are four of them:

```
S contains ( P contains ( J ) )
S contains ( J contains ( P ) )
S contains ( first P then J )
S contains ( first J then P )
```

There are many other candidate designs—for example, an "SP" object that shows directly which suppliers supply which parts and also includes two embedded sets of projects, one for the supplier and one for the part.

There is also a very simple design involving no hierarchies at all, consisting of an "SP" object, a "PJ" object, and a "JS" object.

24 Additional OO Concepts

24.1 Introduction

In this chapter we take a brief look at a somewhat mixed bag of issues. The issues in question are all part of what is usually thought of as the general subject of OO systems, though in fact—despite the overall title of this chapter—every one of those issues is actually quite independent of whether the system is object-oriented or not; there is no reason, for example, why a database programming language should not be used with a relational system (see, e.g., reference [24.2]). The issues are:

- Database programming languages
- Versioning
- Transaction management
- Schema evolution
- Performance considerations

We also consider some more traditional aspects of database management and see how they look in an OO context. Specifically, we examine:

- "Relationships" (i.e., referential integrity)
- General integrity constraints
- "Methods that span classes" (i.e., multi-argument functions)
- *Ad hoc* query

plus a few miscellaneous items. Our discussions will lead to a number of important questions regarding OO systems.

24.2 The Issues

Database Programming Languages

The OPAL examples in the previous chapter illustrate the point that OO systems typically do not employ the *embedded database language* approach found in most (SQL)

relational products today. Instead, the same **integrated language** is used for both database operations and nondatabase operations. To use the terminology of Chapter 2, the host language and the database language are *tightly coupled* in OO systems (in fact, of course, the two languages are one and the same).

Now, it is undeniable that there are many advantages to such an approach [24.2]. One important one is the improved type checking that becomes possible, as discussed in Chapter 19. And reference [22.12] argues that the following is another:

> "With a single unified language, there is no *impedance mismatch* between a procedural programming language and an embedded DML with declarative semantics."

The term **impedance mismatch** refers to the difference in level between today's typical programming languages, which are *record-at-a-time*, and relational languages such as SQL which are *set*-at-a-time. And it is true that this difference in level does give rise to certain practical problems in relational products. However, the solution to those problems is *not* to bring the database language down to the record-at-a-time level (which is what OO systems do)!—it is to introduce set-at-a-time facilities into programming languages instead. The fact that OO languages are record-at-a-time is a throwback to the days of prerelational systems such as IMS and IDMS.

To pursue this latter point a moment longer: It is indeed the case that most existing OO languages are quite procedural or "3GL" in nature. As a consequence, all of the relational set-level advantages are lost; in particular, the system's ability to optimize user requests is severely undermined, which means that—as in prerelational systems—performance issues are largely left to some human user (the application programmer or the DBA). What is more, (a) most installations currently use several distinct host languages anyway, not a "single language" (unified or otherwise); (b) in an OO system, that "single unified language" is likely to be OPAL, C++, or some other OO language that is unfamiliar to most installations.

As an aside, we remark that although many OO languages have been defined, the two most commonly seen in the marketplace are indeed OPAL (or some other dialect of Smalltalk) and C++. C++ in particular [24.6,24.10] is supported by most commercial products and looks like becoming a *de facto* standard (despite the fact that it is not very "OO-pure"). Efforts are also under way to define an official standard version of C++.

Note: Our discussions in this subsection have been concerned specifically with *programming* languages, not with query languages. Indeed, as mentioned in Chapter 22, the idea of *ad hoc* query is somewhat at odds with the "pure" OO idea of object encapsulation; however, most OO systems do provide some kind of query language (usually a derivative of SQL, with a name like **"Object SQL"** or **"OSQL,"** though the reader is cautioned that such languages often do not look much like conventional SQL). Another point of difference *vis-à-vis* relational systems, therefore, is that OO systems typically use two different languages, one for application programming and one for interactive query; thus, they typically do not support the *dual-mode principle* (see Chapter 8, Section 8.8).

We will consider the question of *ad hoc* query further in Section 24.3.

Versioning

Many applications need the concept of multiple **versions** of a given object; examples include software development, hardware design, document creation, and so forth (see, e.g., references [24.4,24.8-24.9]). And some OO systems support this concept directly. Such support typically includes:

- The ability to create a new version of a given object, typically by **checking out** a copy of the object and moving it from the database to the user's private workstation, where it can be kept and modified over a possibly extended period of time (e.g., hours or days)
- The ability to establish a given object version as the current database version, typically by **checking it in** and moving it from the user's workstation back to the database (which might in turn require some kind of mechanism for **merging** distinct versions)
- The ability to **delete** (and perhaps **archive**) obsolete versions
- The ability to interrogate the **version history** of a given object

Note that—as Fig. 24.1 suggests—version histories are not necessarily linear (version V.2 in that figure *branches* into two distinct versions V.3a and V.3b, which are subsequently *merged* to produce version V.4).

Next, because objects are typically interrelated in various ways, the concept of versioning leads to the concept of *configurations*. A **configuration** is a collection of mutually consistent versions of interrelated objects. Configuration support typically includes:

- The ability to **copy** an object version from one configuration to another (e.g., from an "old" configuration to a "new" one)
- The ability to **move** an object version from one configuration to another (i.e., to add it to the "new" configuration and remove it from the "old" one)

Internally, such operations basically involve a lot of pointer juggling—but there are major implications (beyond the scope of this book) for language syntax and semantics in general and *ad hoc* query in particular.

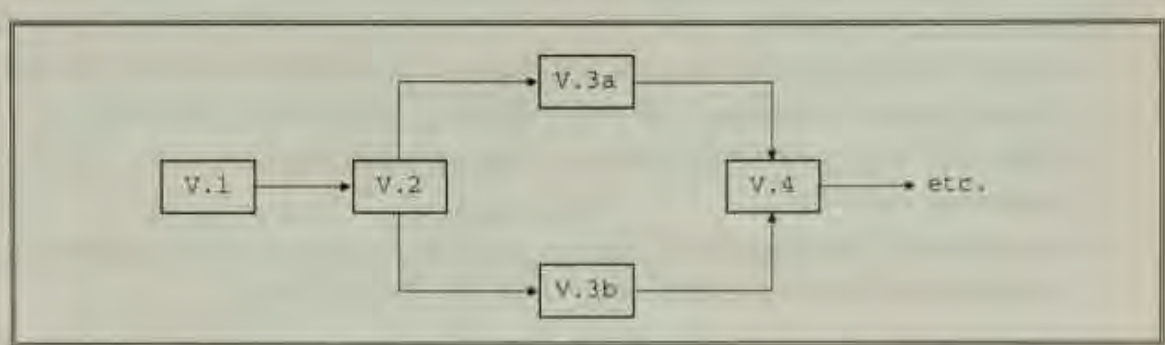

FIG. 24.1 Typical version history

Transaction Management

OO applications often involve complex processing requirements for which the classical transaction management controls described in Chapters 13 and 14 are not well suited. The basic problem is that (as suggested under "Versioning" above) complex transactions might last for hours or days, instead of for a few milliseconds at most as in traditional systems. As a consequence:

1. Rolling back a transaction all the way to the beginning might cause the loss of an unacceptably large amount of work
2. The use of conventional locking might cause unacceptably long delays (waiting for locks to be released)

Point 1 here implies the need to be able to perform *partial* rollbacks; point 2 implies the need for some *nonlocking* concurrency control mechanism. *Long transactions* and *nested transactions* both address these requirements somewhat. We do not discuss these concepts in detail here, but content ourselves with the following observations:

- **Long transactions:** Here the concept of **savepoints** [13.10] and the concept of **sagas** [13.8] are both relevant, since both effectively provide some kind of partial rollback capability. Furthermore, **multiversion** concurrency control [14.1] also looks attractive (especially if the system supports versioning anyway as discussed above), because it implies among other things that:
  - Reads are never delayed (in particular, they are not delayed by any concurrent long transaction)
  - Reads never delay updates (in particular, they do not delay any concurrent long transaction)
  - It is never necessary to roll back a read-only transaction
  - Deadlock is possible only between update transactions
- **Nested transactions:** The nested transaction idea can be thought of as a generalization of savepoints [13.10]. Savepoints allow a transaction to be organized as a sequence of actions that can be rolled back individually; nesting, by contrast, allows a transaction to be organized (recursively) as a *hierarchy* of such actions (see Fig. 24.2). In other words:
  - BEGIN TRANSACTION is extended to support subtransactions (i.e., if BEGIN is issued when a transaction is already running, it starts a *child* transaction)
  - COMMIT TRANSACTION "commits" but only within the *parent scope* (if this transaction is a child)
  - ROLLBACK TRANSACTION undoes work back to the start of this particular subtransaction (including child, grandchild, etc., transactions)

  See the answer to Exercise 13.2 in Chapter 13 for further discussion.

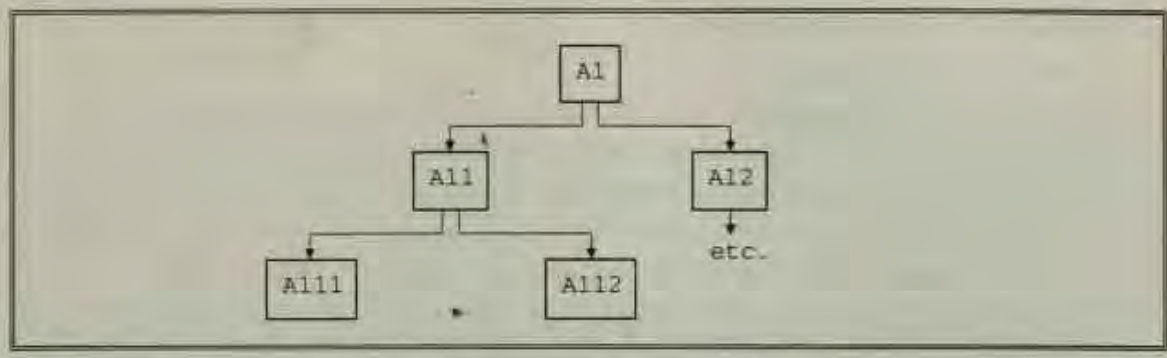

FIG. 24.2 Typical nested transaction

Schema Evolution

Traditional database products typically support only rather simple changes to an existing schema (e.g., the addition of a new attribute to an existing base relation). However, some applications require more sophisticated schema change support, and some OO prototypes have investigated this problem in depth. Note, incidentally, that the problem is actually more complex in an OO environment, because the schema is more complex.

The following taxonomy of possible schema changes is based on one given in a paper on the OO prototype ORION [24.3]:

- Changes to an object class
  1. Instance variable changes
     - add instance variable
     - delete instance variable
     - rename instance variable
     - change instance variable default value
     - change instance variable data type
     - change instance variable inheritance source
  2. Method changes
     - add method
     - delete method
     - rename method
     - change method internal code
     - change method inheritance source
- Changes to the class hierarchy (assuming multiple inheritance)
  - add class *A* to superclass list for class *B*
  - delete class *A* from superclass list for class *B*

- Changes to the overall schema
  - add class (anywhere)
  - delete class (anywhere)
  - rename class
  - partition class
  - coalesce classes

It is not clear how much transparency can be achieved with respect to the foregoing changes, however, especially since view support is typically not included in "the OO model" (see Section 24.3). In fact, the possibility of schema evolution implies a significant problem for OO systems, owing to their record-at-a-time nature. As reference [25.10] puts it: "If the number of indexes changes or the data is reorganized to be differently clustered [see the next subsection below], there is no way for [methods] to automatically take advantage of such changes."

Performance Considerations

Raw performance is one of the biggest objectives of all for OO systems. To quote Cattell [22.2]: "An order-of-magnitude performance difference can effectively constitute a *functional* difference, because it is not possible to use the system at all if performance is too far below requirements" (slightly paraphrased).

Many factors are relevant to the performance issue. Here are a few of them:

- **Clustering:** As we saw in Chapter 18, physical clustering of logically related data on the disk is one of the most important performance factors of all. OO systems typically use the logical information from the schema (regarding type hierarchies, containment hierarchies, or other explicitly declared interobject relationships) as a hint as to how the data should be physically clustered. Alternatively (and preferably), the DBA might be given some more explicit and direct control.
- **Caching:** As suggested under "Versioning" above, OO systems are typically used in a client/server environment, in which users "check out" data from the database and bring it down to their workstation and keep it there for an extended period of time. Caching logically related data at the client site obviously makes sense in such an environment.
- **Swizzling:** The term "swizzling" refers to the process of replacing OID-style pointers by main memory addresses when objects are read into memory (and *vice versa* when the objects are written back to the database). The advantages for applications that process "complex objects" and thus require extensive pointer chasing are obvious.

Reference [22.2] discusses a benchmark called OO1 for measuring system performance on a bill-of-materials database (see also reference [24.5]). The benchmark involves:

1. Random retrieval of 1000 parts, applying a user-defined function to each
2. Random insertion of 1000 parts

3. Random part explosion (up to 7 levels), applying a user-defined function to every part encountered

According to reference [22.2], comparison of an (unspecified) OO product with an (unspecified) state-of-the-art relational product showed up to two orders of magnitude difference in favor of OO—especially on "warm" accesses (after the cache had been populated). However, reference [22.2] is also careful to say that "The differences . . . should *not* be attributed to a difference between the relational and OO models. . . . There is reason to believe that most of the differences can be attributed to [implementation matters]." This disclaimer is supported by the fact that the differences were much smaller when the database was "large" (i.e., when it was no longer possible to accommodate the entire database in the cache).

24.3 Some Questions

As promised in the introduction, we now turn our attention to some aspects of database management that are usually regarded as traditional, see how they look in an OO context, and use them as a basis for raising a number of questions regarding OO technology. The aspects in question are as follows:

- Relationships
- Integrity
- Methods that span classes
- *Ad hoc* query
- Miscellaneous

Relationships

OO products and the OO literature typically use the term "relationships" to refer specifically to those relationships that would be represented by foreign keys in a relational database. But they distinguish:

1. Subclass-to-superclass ("ISA") relationships
2. One-to-many relationships
3. Many-to-many relationships

We discuss each case in turn.

1. Subclass-to-superclass relationships are represented by the **class hierarchy**—e.g.,

   ```
   ... RECTANGLE ISA ( POLYGON ) ...
   ```

   (to use the hypothetical syntax of Chapter 22).

2. One-to-many relationships are represented *either* by a **containment hierarchy**—e.g.,

   ```
   ... DEPT ... ( ... EMPS REF ( SET ( REF ( EMP ) ) ) ) ...
   ```

   or by a mechanism called "inverse variables" (see below).

3. Many-to-many relationships are represented *either* by **object sharing**—e.g., supplier and part objects might both "contain" (and thus share) shipment objects—*or* by "inverse variables" again.

Inverse variables: Consider departments and employees once again. Instead of adopting a containment hierarchy representation, we might decide to define two separate object classes (rather as in a relational database), *and* to include both (a) an EMPS instance variable in each DEPT, corresponding to the set of employees in the department, and (b) an EDEPT instance variable (say) in each EMP, corresponding to the department for this employee:

```
... DEPT ... ( ... EMPS REF ( SET ( REF ( EMP ) ) )
                                    INVERSE EMP.EDEPT ) ...
... EMP  ... ( ... EDEPT REF ( DEPT ) INVERSE DEPT.EMPS ) ...
```

The *reference* instance variable EDEPT and the *references set* instance variable EMPS are said to be **inverses** of each other. Inverse variables obviously need to be kept mutually consistent; this is one aspect of *referential integrity* (as that concept applies to OO systems), and we will return to it in just a moment.

The foregoing example illustrates the one-to-many case. Here is an example of the many-to-many case:

```
... S ... ( ... PARTS REF ( SET ( REF ( P ) ) )
                                    INVERSE P.SUPPS ) ...
... P ... ( ... SUPPS REF ( SET ( REF ( S ) ) )
                                    INVERSE S.PARTS ) ...
```

Here the inverse variables are both *reference set* variables.

One obvious question that arises is the following: How does OO deal with relationships that involve more than two object classes—e.g., suppliers, parts, and projects? The best (i.e., most symmetric) answer to this question, of course, is that an "SPJ" object class is created, in which each SPJ object has an "inverse variables" relationship with the appropriate supplier, the appropriate part, and the appropriate project. Given that creating a new object class is apparently the best approach for "relationships" of degree greater than two, the question arises as to why "relationships" of degree two are not treated in the same way.

Also—regarding inverse variables specifically—why is it necessary to introduce the asymmetry, the directionality, and two different names for what is essentially one thing? For example, the OO analogs of the two relational expressions

```
SP.P# WHERE SP.S# = 'S1'
SP.S# WHERE SP.P# = 'P1'
```

look something like this:

```
S.PARTS.P# WHERE S.S# = 'S1'
P.SUPPS.S# WHERE P.P# = 'P1'
```

(hypothetical syntax, deliberately chosen to avoid irrelevant distinctions).

Referential integrity: As promised, we now take a slightly closer look at OO support for referential integrity (frequently claimed as a strength of OO systems, incidentally).

Various levels of support are possible. The following classification scheme is taken from Cattell [22.2]:

- **No system support:** Referential integrity is the responsibility of the user or user-written methods (just as it was in the original SQL standard, incidentally).
- **Reference validation:** The system checks that all references are to objects of the correct type; however, object deletion might not be allowed (instead, objects might be "garbage collected" when there are no remaining references to them, as in OPAL). As explained in Chapter 23, this level of support is roughly equivalent (but only roughly) to a "DELETE CASCADES" rule for nonshared subobjects within a containment hierarchy and to a "DELETE RESTRICTED" rule for other objects.
- **System maintenance:** Here the system keeps all references up to date automatically (e.g., by setting references to deleted objects to *nil*). This rule is somewhat akin to a "DELETE NULLIFIES" rule in a relational system.
- **"Custom semantics":** "DELETE CASCADES" (outside of the containment hierarchy) might be an example of "custom semantics." Such cases are typically not supported by OO systems at the time of writing, and must therefore be handled by appropriate methods—in other words, by user-written procedural code.

Integrity

We now turn to **general integrity constraints**. Here is an example (taken from Section 16.7): "Suppliers with status less than 20 must not supply any part in a quantity greater than 500." Using the integrity language developed in Chapter 16, this constraint might be expressed as follows in a relational system:

```
CREATE INTEGRITY RULE C95
       IF S.STATUS < 20 AND S.S# = SP.S#
       THEN SP.QTY ≤ 500 ;
```

In an OO system, by contrast, this constraint will almost certainly have to be enforced by *procedural code* (though reference [24.7] does describe a prototype implementation of declarative integrity constraints in an OO system). Such procedural code will have to be included in at least all of the following:

- Method for creating a shipment
- Method for changing a shipment quantity
- Method for changing a supplier status
- Method for assigning a shipment to a different supplier

Points and questions arising:

1. We have obviously lost the possibility of the system determining for itself when to do the integrity checking.
2. How do we ensure that all necessary methods include all necessary enforcement code?
3. How do we prevent the user from (e.g.) bypassing the "create shipment" method

and using the builtin method NEW directly on the shipment object class (thereby bypassing the integrity check)?

4. How do we ensure uniformity of error messages:
   - across all methods involved in enforcing "the same" constraint?
   - across all constraints (e.g., all referential constraints) that bear a strong family resemblance?
5. If the constraint changes, how do we find all methods that need to be rewritten?
6. How do we ensure that the enforcement code is correct?
7. How do we do deferred (commit-time) integrity checking?
8. How do we query the system to find all constraints that apply to a given object or combination of objects?
9. Will the constraints be enforced during load and other utility processing?
10. What about semantic optimization (i.e., using integrity constraints to simplify queries, as discussed in Chapter 18)?

And what are the implications of all of the above for productivity, both during application creation and subsequent application maintenance?

(As a practical matter, incidentally, the SQL/92 standard [8.1] does include declarative support for integrity constraints, and DBMS products, OO or otherwise, are going to have to support that standard. The implications for OO systems are not clear.)

A note on functional dependencies: Functional dependencies (FDs) are a special kind of integrity constraint, and we have seen elsewhere in this book that the system needs to know about FDs in order to behave in a variety of "intelligent" ways. But OO systems provide no way of declaring FDs. In fact, some writers seem to regard such a lack almost as a *virtue*. As we know, FDs are important for database design; however, reference [22.12] says:

> "Complex [data] can be represented directly [in an OO system], without the need to normalize . . ."

Now, it might be true that OO eliminates the "need to normalize," in the sense that it directly supports unnormalized objects (i.e., hierarchies). However, it does not follow that OO automatically eliminates the problems that unnormalized objects cause! The point cannot be overemphasized that "the need to normalize" is not something that is peculiar to relational systems, i.e., something that can be ignored if the system is not relational. On the contrary, the problems that normalization solves are *inherent* problems, and do not go away just because the data structure happens to be something other than relational.

Methods That Span Classes

Consider the education database from Chapter 23 and the query "Find the grade obtained by employee *E* on course *C*" (where *E* and *C* are parameters). Let us suppose a method *M* has been written to implement this query. Is *M* part of the EMPLOYEE

object class or the COURSE object class? Whichever it is, why? How does the user know? What are the implications?

To state matters differently: The idea of a function or method being bundled in with an object class works fine *so long as* the function in question takes just one argument. But as soon as two or more arguments are involved, a degree of arbitrariness inevitably creeps in. In fact, some methods—NEW is an example—are bundled, not with the object class of one of their parameters, but rather with that of their *result*, which seems to introduce yet another element of arbitrariness.

So let us step back for a moment. Why are methods bundled in with an object class anyway? The answer is that they need *privileged access*—access, that is, to the internal representation (i.e., private instance variables) of objects of the class in question. (Access to other objects, by contrast, is by means of the usual GET and SET methods.)

But we already have a mechanism for dealing with privileged access, namely the **security** mechanism.* Would it not be better to unbundle methods from object classes, and instead use the security mechanism to control which methods are allowed to access the internals of which objects? Such an approach would have the advantages that:

- Methods could be permitted to access the internals of any number of objects instead of being limited to just one.
- The asymmetry, arbitrariness, and awkwardness inherent in the bundling scheme would be eliminated.
- In particular, the *ad hoc* distinction between private and protected instance variables would be unnecessary.

In other words, it is the opinion of this writer that the bundling of methods with object classes should not be regarded as a fundamental component of "the OO model." Please note that this is not a criticism of methods as such, only of the bundling aspect of such methods.

Ad Hoc Query

We have suggested a couple of times already that OO systems might have some difficulty over *ad hoc* query. As we put it in Chapter 22, ". . . if it is really true that predefined methods are the only way to manipulate objects, then *ad hoc* query is impossible!—unless objects are designed in accordance with a certain very specific discipline." For example, if the only methods defined for DEPT objects are HIRE_EMP, FIRE_EMP, and CUT_BUDGET, then even a query as simple as "Who is the manager of the programming department?" cannot be handled.

Careful consideration of examples such as the foregoing leads to the following conclusion. If we are indeed to be able to permit *ad hoc* queries against objects of some given class, then we must do the following two things:

---

* Note that we are talking about OO *database* systems specifically here. OO programming systems typically did not have a security mechanism, and this historical lack probably accounts for the current situation.

1. Define and maintain a "collection" that is the set of all currently existing objects of that class. This collection will serve as a defining scope for queries, much as base relations do in a relational system.
2. Define the public interface for that class in such a way as to expose *one candidate representation* of the objects to the user.

For example, consider the classes POINT and LINESEG (line segment). If we define GET methods for the start and end points of line segments, and GET methods for the *x*- and *y*-coordinates of points—and if we permit references to such methods to be nested within scalar expressions in arbitrary ways—and if we do indeed define and maintain a collection that is the set of all line segments, and another collection that is the set of all points—then we can indeed define an *ad hoc* query language that will permit the user to ask arbitrary unplanned queries against objects of those classes. For example:

- Find all line segments starting at the origin (0,0)
- Find all line segments of length one unit
- Find the midpoint of a specified line segment

(and so on). *Note very carefully that the fact that line segments might internally be represented by MIDPOINT, LENGTH, and SLOPE, and points by polar coordinates R and THETA, is neither here nor there.*

Points arising:

1. GET methods are sufficient for query *per se*, of course. SET methods are needed to permit arbitrary updates.
2. There is nothing wrong with the idea of exposing more than one candidate representation for a given class. In the case of class POINT, for example, we could define methods GET_*x* and GET_*y and* methods GET_*r* and GET_θ, thereby making queries that are expressed in terms of polar coordinates just as easy to formulate as ones expressed in terms of Cartesian coordinates. In effect, we would be telling the user that each point has an *x* property and a *y* property and an *r* property and a θ property. However, we could not pretend that these properties were all independent of one another; we would have to explain that, e.g., changing the value of *x* or *y* will have the side-effect of changing both *r* and θ, in general. (Furthermore, we would have to explain the exact nature of the relationship between (*x*,*y*) and (*r*,θ), so that the details of those side-effects would be explicable and predictable.)

Note: We do not mean to suggest by the foregoing that the various GET and SET methods should be the *only* methods defined for a given class. At the very least, there needs to be an appropriate constructor function (NEW or some analog thereof) to create new instances of the class, also a corresponding "destructor" function to destroy such instances again. In addition, there almost certainly need to be methods for comparing class instances, assigning class instances, etc.

Miscellaneous

We close this section with some comments and questions regarding a few miscellaneous topics: views, the catalog, missing information, and closure.

- **Views:** What is the analog of views in an OO system?

  Some OO systems do support **derived** (public) **instance variables**—e.g., the instance variable AGE might be derived by subtracting the value of the instance variable BIRTHDATE from the current date—but such a capability falls far short of a full view mechanism. *Note:* This question is related to the question of derived objects in general (see Chapter 25). See also reference [24.1], which describes some research ideas for view support in an OO system.

- **The catalog:** Where is the catalog in an OO system? What does it look like? Are there any standards?

  Incidentally, this is as good a place as any to point out that there is really a **difference in kind** between an OO DBMS and a relational DBMS. A relational DBMS comes *ready for use:* As soon as the system is installed, users can start building databases, writing applications, running queries, and so on. An OO DBMS, by contrast, is best thought of as a kind of *DBMS construction kit:* It provides a set of building blocks—object library maintenance tools, method compilers, etc.—that can be used by technical staff (i.e., database professionals) to *construct* a DBMS in the more familiar sense of the term. In particular, those database professionals will have to construct an appropriate *catalog* for the objects and methods they create. What is more, that constructed DBMS will be *application-specific:* It might, for example, be tailored for a certain set of CAD/CAM applications, but be essentially useless for, e.g., medical applications.

- **Missing information:** Missing information is far from being a fully solved problem in relational systems (see Chapter 20). But OO systems mostly seem to ignore the problem altogether; at best, the evidence suggests that the OO community lags some way behind the relational community in this regard. For example: "Is *nil* a member of every [class]?" [22.12].

- **Closure:** What is the OO analog of the relational closure property? *Note:* This question is actually an extremely important one. We will revisit it in Chapter 25.

24.4 Summary

In this chapter we have concerned ourselves with the following major topics.

- *Database programming languages:* We explained the **impedance mismatch** problem, and claimed that the solution to that problem was not to bring the database language down to the record-at-a-time level, but rather to raise the programming language to the set-at-a-time level. Indeed, the fact that OO languages are so procedural is a severe disadvantage of those languages; in particular, it seriously undermines optimizability.

- *Versioning:* We briefly described the ideas of **version** and **configuration** support and the basic notions of **checking out** database objects and subsequently **checking in** those objects again.
- *Transaction management:* OO transactions might last for hours or days, instead of just a few milliseconds. As a result, support for **long** and **nested transactions** becomes desirable. We briefly considered the use of **multiversion concurrency control** in this connection and summarized some of its advantages.
- *Schema evolution:* We presented a taxonomy of possible schema changes, and pointed out in passing that the schema evolution problem is actually more complex for an OO system, because (a) the schema itself is more complex, (b) "the OO model" typically does not include any view support, and (c) OO code is typically record-at-a-time.
- *Performance considerations:* We stressed the point that performance is one of the biggest objectives of all for OO systems, and briefly considered certain performance-related issues, *viz.* **clustering, caching,** and **pointer swizzling**. We also mentioned the **OO1 benchmark**.
- *Relationships:* The term "relationships" is typically used in OO contexts to refer specifically to the OO analog of foreign key relationships in the relational world. However, **subclass-to-superclass** relationships, **one-to-many** relationships, and **many-to-many** relationships are typically all treated differently. In particular, we discussed the use of **inverse variables** and various possible **levels** of referential integrity support—none at all, "reference validation," system maintenance, and "custom semantics." Where possible, we related these cases to their relational analogs.
- *Integrity:* We were very critical of the fact that OO systems typically handle integrity constraints procedurally (via methods), not declaratively.
- *Methods that span classes:* We were also critical of the fact that OO systems bundle methods in with object classes, especially as the choice of which particular class the method is to be bundled with is often arbitrary. We suggested that methods in fact *not* be so bundled, and that suitable extensions to the system's **security** mechanism would be a preferable means of controlling which methods have access to the internals of which objects.
- *Ad hoc query:* We proposed a particular discipline that would allow OO systems to provide full *ad hoc* query support. The key component of that discipline consisted in exposing *one candidate representation* of objects to the user (through appropriate GET methods).
- *Miscellaneous:* Finally, we offered a few comments regarding **views,** the **catalog, missing information,** and **closure**. In particular, we pointed out that there is really a *difference in kind* between an OO DBMS and a relational DBMS, inasmuch as a relational DBMS comes ready for use, whereas an OO DBMS, by contrast, is more of a *construction kit* for building application-specific DBMSs.

To pursue this last point a little further: We could speculate that the difference in kind referred to reflects a difference in origins between the two disciplines. It appears to this writer that OO database technology most likely grew out of a desire on the part of OO application programmers—for a variety of application-specific reasons—to keep their application-specific objects in persistent memory. That "persistent memory" could then certainly be regarded as a database, but (a) it was indeed application-specific, and as a consequence (b) there was little perceived need to be able to perform *ad hoc* queries against it. Thus, the notions that

1. Such a database might be shared among several applications, and hence that
2. The data should be subject to certain application-independent integrity constraints, and also that
3. The data should likewise be subject to certain security constraints,

all surfaced later, after the basic idea of storing objects in a database was first conceived, and thus all constitute "add-ons" to the original "OO model."

Now, the foregoing speculations might be a little wide of the mark, but they do seem to this writer to explain some of the differences in emphasis that can be observed in OO database systems *vis-à-vis* traditional database systems.

Exercises

24.1 What do you understand by the terms *version* and *configuration?*

24.2 Why are *long* transaction support and *nested* transaction support desirable?

24.3 Sketch an implementation mechanism for multiversion concurrency control.

24.4 Consider the performance factors discussed briefly in Section 24.2. Are any of them truly OO-specific? Justify your answer.

24.5 OO systems typically support integrity constraints in a *procedural* fashion, via methods; the main exception is that *referential* constraints are typically supported (at least in part) *declaratively*. What are the advantages of procedural support? Why do you think referential constraints are handled differently?

24.6 Explain the concept of *inverse variables*.

24.7 Consider the *schema evolution* taxonomy presented in Section 24.2. Which items in that taxonomy have counterparts in a relational system? What are those counterparts?

24.8 What are the implications of schema evolution for existing data?

24.9 Investigate any OO system that might be available to you. What programming language does that system support? Does it support a query language? If so, what is it? In your opinion, is it more or less powerful than conventional SQL? Does the system support versioning and configurations? If so, what is involved on the part of the user in managing versions and in interrogating version histories? Does the system support long or nested transactions? Does the system have any schema evolution capabilities? What does the catalog look like? How does the user interrogate the catalog? Is there any view support? If so, how extensive is it? (e.g., what about view updating?) How is missing information handled?

References and Bibliography

See also the references in Chapters 22 and 23.

24.1 Serge Abiteboul and Anthony Bonner. "Objects and Views." Proc. ACM International Conference on Management of Data, Denver, Colo. (May 1991).

Proposes a view mechanism for OO systems. The proposal appears to be the first to take *behavior* (methods) into account.

24.2 Malcolm P. Atkinson and O. Peter Buneman. "Types and Persistence in Database Programming Languages." *ACM Comp. Surv. 19*, No. 2 (June 1987).

This paper is recommended as the best starting point for reading in the area of database programming languages in general.

24.3 J. Banerjee *et al.* "Data Model Issues for Object-Oriented Applications." *ACM TOOIS (Transactions on Office Information Systems) 5*, No. 1 (March 1987). Republished in M. Stonebraker (ed.), *Readings in Database Systems*. San Mateo, Calif.: Morgan Kaufmann (1988). Also republished in reference [22.16].

24.4 Anders Björnerstedt and Christer Hultén. "Version Control in an Object-Oriented Architecture." In reference [22.9].

24.5 R. G. G. Cattell and J. Skeen. "Object Operations Benchmark." *ACM TODS 17*, No. 1 (March 1992).

24.6 Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Reading, Mass.: Addison-Wesley (1990).

24.7 H. V. Jagadish and Xiaolei Qian. "Integrity Maintenance in an Object-Oriented Database." Proc. 18th International Conference on Very Large Data Bases, Vancouver, Canada (August 1992).

Proposes a declarative integrity mechanism for OO systems, showing how an integrity constraints compiler can incorporate the necessary integrity checking code into methods for the appropriate object classes.

24.8 R. H. Katz and E. Chang. "Managing Change in a Computer-Aided Design Database." Proc. 13th International Conference on Very Large Data Bases, Brighton, UK (September 1987).

24.9 John F. Roddick. "Schema Evolution in Database Systems—An Annotated Bibliography." *ACM SIGMOD Record 21*, No. 4 (December 1992).

24.10 Bjarne Stroustrup. *The C++ Programming Language*. Reading, Mass.: Addison-Wesley (1986).

Answers to Selected Exercises

24.3 The following brief description is taken from reference [21.14]. First of all, the system must keep:

1. For each data object, a stack of committed versions (each stack entry giving a value for the object and the ID of the transaction that established that value; i.e., each stack entry essentially consists of a pointer to the relevant entry in the log). The stack is in reverse chronological sequence, with the most recent entry being on the top.
2. A list of transaction IDs for all committed transactions (the *commit list*).

When a transaction starts executing, the system gives it a private copy of the commit list.

Read operations on an object are directed to the most recent version of the object produced by a transaction on that private list. Update operations, by contrast, are directed to the actual current data object (update/update conflict testing is thus still necessary). When the transaction commits, the system updates the commit list and the data object version stacks appropriately.

24.4 The performance factors discussed were *clustering*, *caching*, and *pointer swizzling*. All of these techniques are applicable to any system that provides a reasonable level of data independence; they are thus not truly "OO-specific." In fact, the idea of using the logical database definition to decide what physical clustering to use, as some OO systems do, could be seen as potentially *undermining* data independence. *Note:* It should be pointed out too that another very important performance factor, namely **optimization,** typically does *not* apply to OO systems.

24.5 It is the opinion of this writer that declarative support, if feasible, is *always* better than procedural support (for everything, not just for integrity constraints). In a nutshell, declarative support means that the system does the work instead of the user. This is why relational systems support declarative queries, declarative view definitions, declarative cursor definitions (see Chapter 8), declarative integrity constraints, and so on.

25 Toward an OO/Relational Rapprochement

25.1 Introduction

We have now examined all of the major aspects of OO systems. We began by claiming (in Chapter 22) that today's relational products are inadequate in a variety of ways, and we gave an example ("Find all rectangles that overlap the unit square") in support of that claim. And we went on to say that some of the new features that seem to be needed in DBMSs have existed for many years in OO programming languages, and hence the idea of **OO database systems** is a natural one to investigate. *BUT*—and it is a very big "but"—it does not follow that OO database systems can or will replace relational systems. On the contrary, we should be looking for a *rapprochement* between the two technologies—that is, a way of marrying the two technologies together, so as to get the best of both worlds. The purpose of this final chapter is to investigate the possibility of such a rapprochement.

Now, it is this writer's strong opinion that any such rapprochement should be firmly founded on the relational model (which is after all the foundation of modern database technology in general, as explained in Part II of this book). That is, what we want is for relational systems to evolve to incorporate the features—or, at least, the good features—of OO. We do *not* want to discard relational systems entirely, nor do we want two totally disparate systems, relational and OO, existing side by side as equals but (perhaps) competitors. And this opinion is shared by many other writers, including in particular the authors of the "Third Generation Database System Manifesto" [25.10], who state categorically that **third-generation DBMSs must subsume second-generation DBMSs**—where "second-generation DBMSs" basically means relational systems, and "third generation DBMSs" means whatever comes next. The opinion is apparently *not* shared, however, by some of the early OO products, nor by certain OO researchers. Here, for example, is a typical quote:

> "Computer science has seen many generations of data management, starting with indexed files, and later, network and hierarchical DBMSs ... [and] more recently relational DBMSs . . . Now, we are on the verge of another generation of database

systems . . . [that] provide *object management*, [supporting] much more complex kinds of data" [25.4].

Here the writer is clearly suggesting that just as relational systems displaced the older hierarchic and network systems, so OO systems will displace relational systems in exactly the same kind of way.

The reason we disagree with this position is that (to quote reference [3.2]) *relational really is different*. It is different because it is not *ad hoc*. The older, prerelational systems were *ad hoc*; they might have provided solutions to certain important problems of their day, but they did not rest on any solid theoretical foundation.* Unfortunately, relational advocates—this writer included—did themselves a major disservice in the early days when they argued the relative merits of relational and prerelational systems; such arguments were necessary at the time, of course, but they had the unlooked-for effect of reinforcing the idea that relational and prerelational DBMSs were essentially the same kind of thing. And this mistaken idea in turn supports the position of the quotation above from reference [25.4]—to wit, that OO is to relations as relations were to hierarchies and networks.

In what follows, therefore, we take it as an axiom that what we want to do is enhance relational systems to incorporate the good features (but not the bad features!) of OO. To repeat, we do *not* want to discard relational systems entirely. It would be a great pity to walk away from 25 years of solid relational research and development.

The plan of the chapter is as follows. Section 25.2 reviews the features of OO, in an attempt to separate the good from the bad. Sections 25.3 and 25.4 then discuss ways in which the good features might be added to relational systems; Section 25.3 concentrates on domains and Section 25.4 on relations *per se*. Section 25.5 describes the benefits of a true rapprochement. Section 25.6 offers a summary.

25.2 A Review of "the OO Model"

Below is a summary of the principal features of "the OO model," with a subjective assessment as to which features are essential, which are "nice to have" but not essential, which are bad, etc.

- **Objects:** Objects themselves, both *mutable* (variables) and *immutable* (values), are clearly essential.

---

* What about OO? Is OO *ad hoc*? The following quote from "The Object-Oriented Database System Manifesto" [25.1] is interesting in this regard: "With respect to the specification of the system, we are taking a Darwinian approach: We hope that, out of the set of experimental prototypes being built, a fit [object-oriented] model will emerge. We also hope that viable implementation technology for that model will evolve simultaneously."

- **Object IDs:** Unnecessary, and in fact undesirable (see the "Conclusion" subsection at the end of Section 25.3).
- **Object classes** (meaning **types**): Essential. The associated notion of a *constructor function*—which among other things effectively provides a way of writing a *literal value* of the relevant type—is essential too.
- **Encapsulation:** See the subsection "A note on encapsulation" below.
- **Instance variables:** First of all, *private* and *protected* instance variables are by definition merely implementation matters and are thus not relevant to the definition of an abstract, user-oriented model, which is what we are concerned with here. Second, *public* instance variables do not exist in a pure OO system and are thus also not relevant. Third, *derived* instance variables are a special case of public instance variables and are thus *a fortiori* irrelevant also. We conclude that instance variables can be ignored; objects should be manipulable solely by methods.
- **Containment hierarchy:** We have already explained in Chapter 22 that in our opinion the containment hierarchy concept is misleading and in fact an illusion. Objects are really *tuples*, not hierarchies.
- **Methods:** This concept is essential, of course (although we would prefer the more conventional terms "function" or "operator" in place of the term "method"). Bundling method definitions with object classes is *not* essential, however, and leads to several problems. We would prefer to use the *security* mechanism to control which functions have access to the internals of which objects.
- **Messages:** Again, the concept is essential, though we would prefer to replace the term by the more conventional term "call" or "invocation"—always recognizing that such calls or invocations are not necessarily synchronous.
- **Class hierarchy:** Essential. So too is the related notion of *inheritance*, although—since we have no notion of (public) instance variables—there is no need to distinguish between *structural* and *behavioral* inheritance (all inheritance is by definition behavioral). **Polymorphism** is thus essential too, with its likely implication of *run-time binding* (discussed in Chapter 19)— though this latter notion is an implementation matter and should not be regarded as part of "the OO model" as such.
- **Multiple inheritance:** Secondary. (Actually this concept is likely to prove quite important, but it seems to this writer that there are some significant conceptual and definitional issues to be resolved before we can include such a feature in a well-defined abstract "OO model" with any significant degree of confidence. More study is required.)
- **Class *vs.* instance *vs.* collection:** The distinctions are essential, of course, but not really exclusive to OO (the *concepts* are distinct, and that is really all that needs to be said).
- **Database programming language:** Nice to have, but not exclusive to OO.

- **3GL nature:** A giant step backward.
- **Versions:** Nice to have, but not exclusive to OO. The same goes for the related concept of **configurations**.
- **Long and nested transactions:** Nice to have, but not exclusive to OO.
- **Schema evolution:** Nice to have, but not exclusive to OO.

And here is a list of features that "the OO model" typically does *not* support:

- ***Ad hoc* query:** As explained in Chapter 24, early versions of "the OO model" typically did not include *ad hoc* query capabilities, because (a) the need for such capabilities was not so pressing in the environment in which OO systems originated, and (b) such capabilities seem to imply *either* that objects must be designed according to some discipline such as that outlined in Section 24.3 *or* that queries will have to break encapsulation.
- **Closure:** Generic operators such as (object-level) union, join, projection, etc., are not supported. Derived objects in general are not supported.
- **Symmetric exploitation:** Not supported. Each and every access request involves a distinct, precoded, procedural method.
- **Foreign keys:** The "OO model" has several different mechanisms for dealing with referential integrity, none of which is the same as the relational model's more uniform foreign key mechanism. Such matters as "DELETE RESTRICTED" and "DELETE CASCADES" are typically left to procedural code (probably methods, possibly application code).
- **Declarative integrity constraints:** Currently not supported (for reasons not unrelated to the reasons *ad hoc* query is typically not supported).
- **Views:** Currently not supported (for reasons not unrelated to the reasons *ad hoc* query is typically not supported).
- **Catalog:** Must be built by the professional staff whose job it is to tailor the OO DBMS for whatever application it has been installed for. There is no universal "common catalog" (nor was there for relational systems, of course, prior to SQL/92).

To summarize, the good (and essential) features of the "OO model"—i.e., the ones we would like relational systems to support—are as follows:

- **Objects** and **object classes,** plus corresponding constructor functions (returning *objects*, however, not OIDs—see Section 25.3) and other access and maintenance **functions,** together with a (necessarily) clear distinction among classes, instances, and collections
- **Class hierarchies,** with (behavioral) **inheritance,** and therefore *polymorphism* and run-time binding (though this latter is not part of the model as such)

A Note on Encapsulation

We have said that the system should support encapsulated objects, i.e., objects that are accessible *only* via appropriate access functions (there should be no "public instance variables" in those objects at all). The advantage of such encapsulation is basically that it provides **data independence,** as explained in Chapter 22. (In fact, OO systems are sometimes said to provide a *particularly strong form* of data independence [22.7]. It is important to understand, however, that—at least in principle—OO systems cannot provide any more data independence than relational systems can. For example, there is absolutely no reason—at least in principle—why a base relation called POINT, with Cartesian coordinate attributes X and Y, should not map to a stored relation that represents points by their polar coordinates R and θ instead. As we remarked in Chapter 3, however, today's relational products unfortunately do not provide nearly as much data independence as we would really like, or as much as relational systems are theoretically capable of.)

Note, however, that it is possible to take the encapsulation idea too far. There will always be a need to access the data in ways that were not foreseen when the database was originally created, and the notion of having to write new procedural code every time a new data access requirement arises is simply not acceptable. Indeed, such was exactly the state of affairs with prerelational systems, and it was a major contributor to their downfall.

Thus we see that the system should additionally support "objects" that are *not* encapsulated but expose their "instance variables" for all the world to see. And—to jump ahead of ourselves for a moment—those nonencapsulated "objects" are, precisely, **tuples** in **relations** (see Section 25.3). Note that, at least in principle, nonencapsulated relations can provide just as much data independence as can encapsulated objects (as explained above); note too that, again in principle, the ideas of type hierarchies, inheritance, etc., apply at least as well to nonencapsulated relations as they do to encapsulated objects (as explained in Chapter 19). And, of course, relations can be accessed by the well-known, builtin, generic operators of the relational algebra instead of just by methods that are specific to the particular relation in question, thus satisfying the "unforeseen requirements" objective.

25.3 Domains *vs.* Objects

In this section we argue that the key to the desired rapprochement is the relational notion of **domains** (indeed, we stated this as our position on the subject in Chapter 19). For consider:

- The fundamental construct in OO systems is the **object class,** which is basically a user-defined, encapsulated data type of arbitrary internal complexity.
- The fundamental construct in relational systems—mostly not implemented, unfor-

tunately, in today's relational products—is the **domain,** which is (again) basically a user-defined, encapsulated data type of arbitrary internal complexity.

In other words, a domain and an object class are *the same thing!* Thus, a relational system that implemented domains properly would be able to do all the things that (it is claimed) OO systems can do and relational systems cannot.

Let us review the principal properties of domains as discussed in Chapter 19. We showed in that chapter:

- How domains really are data types of arbitrary complexity
- How the "scalar" values in such a domain can therefore have arbitrarily complex internal structure
- How those values are encapsulated, however, and can be operated on only by predefined functions

We also discussed:

- The fact that the function definitions are typically *not* bundled with the domain definitions
- The relevance of the "strong typing" concept from programming languages
- The ideas of inheritance and polymorphism as they apply to domains

Indeed, domains and object classes really are one and the same. Thus, domains (and therefore relational attributes) can contain absolutely anything—arrays, lists, stacks, documents, photographs, maps, blueprints, etc., etc. Another way of saying this is: **Domains encapsulate, relations don't.**

The Rectangles Problem Revisited

So now we can give a "good" solution to the rectangles problem. First of all, we define a domain of rectangles ("we" here probably meaning the DBA):

```
CREATE DOMAIN RECTANGLE REP ( ... ) ;
```

We assume here that the representation ("REP") is in terms of one of the newer storage structures (quadtrees, grid files, R-trees, etc.) that are specifically intended to support spatial data such as geometric figures efficiently—see Appendix A, references [A.48–A.59].

Next we define a "constructor" function for creating new rectangles (i.e., for converting a set of four floating-point numbers into a rectangle):

```
CREATE FUNCTION RECTANGLE
     ( A FLOAT, B FLOAT, C FLOAT, D FLOAT )
       RETURNS ( RECTANGLE )
   AS BEGIN ;
         DECLARE R RECTANGLE ;
         R.X1  :=  A ;  R.X2  :=  B ;
```

```
        R.X1 := C ; R.Y2 := D ;
        RETURN ( R ) ;
    END ;
```

Here we assume that R.X1 refers to the *x*-coordinate of the bottom left-hand corner of rectangle R, etc., and that a rectangle can be defined by specifying these four coordinates. Note, incidentally, that we do *not* bundle the RECTANGLE function with the RECTANGLE "object class" (domain).

We also define a function to test two rectangles to see whether they overlap:

```
CREATE FUNCTION OVERLAP
     ( R1 RECTANGLE, R2 RECTANGLE )
       RETURNS ( BOOLEAN )
   AS BEGIN ;
       ... code ... ;
       END ;
```

The *code* here is low-level implementation code, probably written in a language like C, to implement the *efficient* (short) form of the "overlaps" test against the *efficient* (R-tree or whatever) storage structure.

Now an SQL user can create a base table with a column defined on the rectangles domain:

```
CREATE TABLE RECTANGLES
     ( RECTID ... , R RECTANGLE ... , ... ,
       UNIQUE ( RECTID ) ,
       UNIQUE ( R ) ) ;
```

And the query "Find all rectangles that overlap the unit square" now becomes simply:

```
SELECT ...
FROM   RECTANGLES
WHERE  OVERLAP ( R, RECTANGLE ( 0, 1, 0, 1 ) ) ;
```

Note that the expression RECTANGLE(0,1,0,1) in this query effectively constitutes a kind of "rectangle literal."

Conclusion

The system should support *relations*, with the "extension" (actually not an extension at all) that *domains* are properly supported and can be user-defined data types of arbitrary complexity. Now, we can call the elements of such domains *objects* if we like (they are totally encapsulated); note, however, that at the relational level **there is no need for OIDs**—the relational model requires relations to contain *actual objects* (at least conceptually), not pointers to objects. Thus, for example, the RECTANGLE function (which is the relational analog of "NEW" for rectangles) does not return a rectangle OID, it returns (at least conceptually) *an actual rectangle*.

As for the "class *vs.* instance *vs.* collection" distinctions:

- The *class* is the domain.
- An *instance* is an element of the domain. So far as the database is concerned,

such instances exist only within attribute positions within tuples within relations; however, we can create new "literal" instances at any time for use as arguments to whatever operations or functions are defined for the domain in question.

- A *collection* is the collection of values of any attribute defined on the domain. As for sharing the very same instance among several collections, the *effect* of such sharing can be achieved by means of foreign keys; actually placing two copies of the same instance in two distinct base relations is *not* sharing in the same sense (and in fact is typically contraindicated by the principles of further normalization). *Note:* The very same instance can of course be shared among exactly one base relation and any number of other (named) relations if those other relations are all *views*.

To sum up: Relational vendors should do all in their power to extend their systems to include proper domain support. Indeed, an argument can be made that the whole reason that OO systems look attractive is precisely because the relational vendors to date have failed to support the relational model adequately. But this fact should not be seen as an argument for abandoning relational systems entirely (or at all!).

25.4 Relations *vs.* Objects

In the previous section we equated object classes and domains. Many products and prototypes, however, are equating object classes and *relations* instead.* In this section we argue that this latter equation is a serious mistake.

We begin by considering a simple example. Here is a slightly simplified version of one of the very first CREATE OBJECT CLASS examples from Chapter 22:

```
CREATE OBJECT CLASS EMP
PUBLIC ( EMP#  CHAR,
         ENAME CHAR,
         SAL   NUMERIC ) ... ;
```

And here is a relational (SQL) analog:

```
CREATE TABLE EMP
       ( EMP#   CHAR,
         ENAME  CHAR,
         SAL    NUMERIC ) ... ;
```

* Regarding the general question of whether any two concepts *A* and *B*, such as relations and object classes, are "the same" and can thus be equated, there is a simple general principle—previously articulated by this writer in reference [4.12], and well worth repeating here—that can usefully be applied: Simply ask yourself "Is every instance of *A* an instance of *B*?" and, conversely, "Is every instance of *B* an instance of *A*?" Only if the answer is *yes* to both of these questions is it true that *A* and *B* are identical.

It is very tempting to equate the two! And many systems have done exactly that, including Iris [25.3,25.7], POSTGRES [25.13-25.16], UniSQL [25.11], and others. UniSQL, for example, supports an extended form of SQL called SQL/X, which we illustrate by means of the following example (based on one in the SQL/X user's manual [25.11]). We start with a conventional, self-explanatory SQL CREATE TABLE statement:

```
CREATE TABLE EMP
     ( EMP#        CHAR(5),
       ENAME       CHAR(20),
       SAL         NUMERIC,
       HOBBY       CHAR(20),
       WORKS_FOR   CHAR(20) ) ;
```

Two points arise immediately:

1. SQL/X does not currently support the SQL PRIMARY and FOREIGN KEY clauses. Furthermore, since the intent is to equate CREATE TABLE and "CREATE OBJECT CLASS"—as a matter of fact, SQL/X allows the keyword TABLE in CREATE TABLE to be replaced by the keyword CLASS—the implications of adding such support are not clear, since primary and foreign keys are not part of the object class concept.
2. SQL/X also does not support domains. Indeed, those who espouse the "relation = object class" equation typically never even mention domains. This omission is probably due to the widespread failure on the part of the relational vendors to implement domains, which in turn is due to the failure of the original SQL language to include any domain support in the first place.

Anyway, to get back to the example: The first SQL/X extension is to add **composite column** support. Thus, e.g., we can replace the original CREATE TABLE by the following (refer to Fig. 25.1):

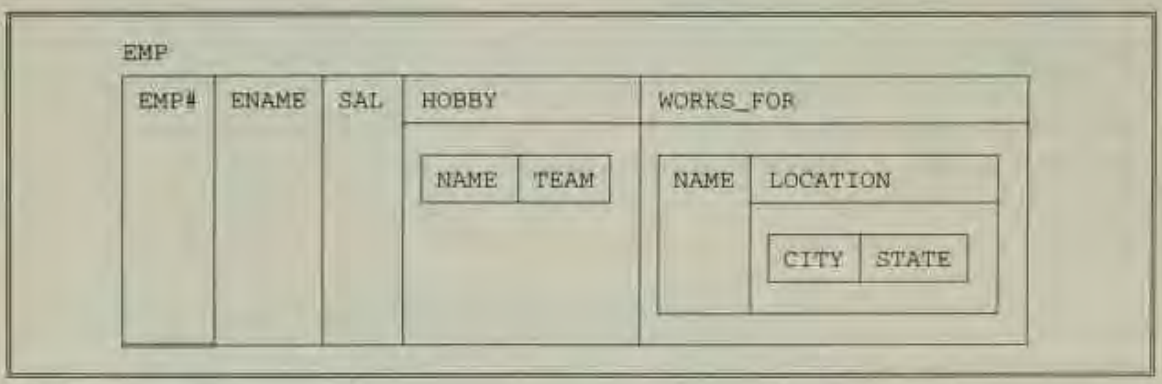

FIG. 25.1 SQL/X table with composite columns

```
CREATE TABLE EMP
     ( EMP#        CHAR(5),
       ENAME       CHAR(20),
       SAL         NUMERIC,
       HOBBY       ACTIVITY,
       WORKS_FOR   COMPANY ) ;

CREATE TABLE ACTIVITY
     ( NAME  CHAR(20),
       TEAM  INTEGER ) ;

CREATE TABLE COMPANY
     ( NAME        CHAR(20),
       LOCATION    CITYSTATE ) ;

CREATE TABLE CITYSTATE
     ( CITY        CHAR(20),
       STATE       CHAR(2) ) ;
```

Explanation:

- The column HOBBY in table EMP is declared to be of type ACTIVITY. ACTIVITY in turn is a table of two columns, NAME and TEAM, where TEAM represents the number of players in the corresponding team—e.g., a possible "activity" might be (Soccer,11). Each HOBBY value is thus actually an ordered pair of values, a NAME value and a TEAM value. (More precisely, it is an ordered pair of values that currently appears in the ACTIVITY table.) Incidentally, those HOBBY values are *not* encapsulated (Fig. 25.1 notwithstanding); thus, e.g., a "path expression" of the form EMP.HOBBY.NAME is legal (see below).
- Similarly, column WORKS_FOR in table EMP is declared to be of type COMPANY, and COMPANY is also a table of two columns, one of which is defined to be of type CITYSTATE, which is another two-column table, and so on.

In other words, tables ACTIVITY, COMPANY, and CITYSTATE are all considered to be *types* as well as tables (the same is true for table EMP also, of course). Thus, the first SQL/X extension is roughly analogous to allowing objects to contain other objects (it is SQL/X's version of support for the containment hierarchy).

The next extension is to add **table-valued columns**. Suppose, for example, that employees can have an arbitrary number of hobbies, instead of just one (see Fig. 25.2):

```
CREATE TABLE EMP
     ( EMP#        CHAR(5),
       ENAME       CHAR(20),
       SAL         NUMERIC,
       HOBBIES     SET OF ( ACTIVITY ),
       WORKS_FOR   COMPANY ) ;
```

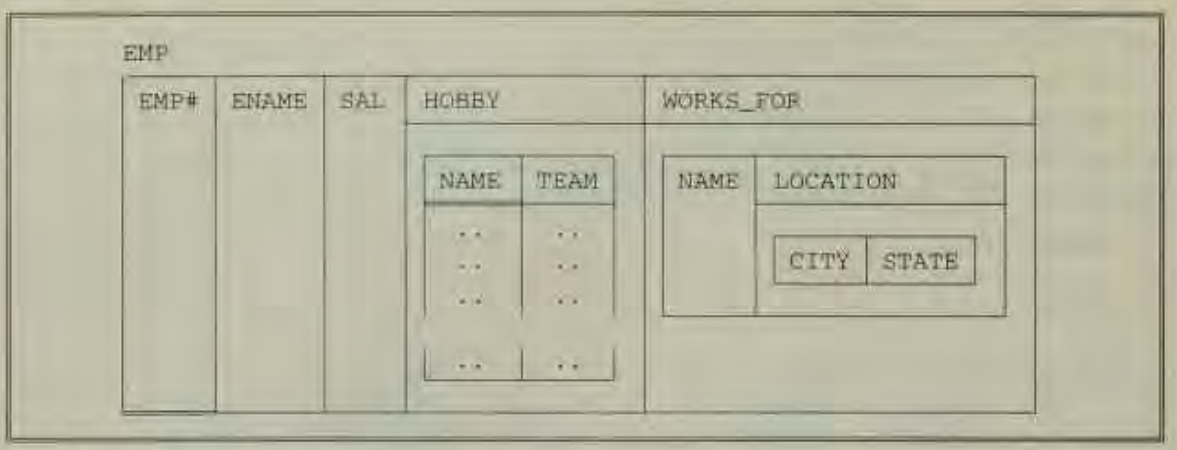

FIG. 25.2 SQL/X table with a table-valued column

Explanation:

- The HOBBIES value within any given row of table EMP is now (conceptually) a *set* of zero or more (NAME,TEAM) pairs from the ACTIVITY table. This second extension is thus roughly analogous to allowing objects to contain "aggregate" objects—a more complex version of the containment hierarchy. *Note:* The aggregate object types supported by SQL/X are *sets*, *sequences*, and *multisets* (bags).

  The third extension is to permit tables to have associated **methods**. For example:

```
CREATE TABLE EMP
     ( EMP#        CHAR(5),
       ENAME       CHAR(20),
       SAL         NUMERIC,
       HOBBIES     SET OF ( ACTIVITY ),
       WORKS_FOR   COMPANY )
METHOD RETIREMENT_BENEFITS ( ) : NUMERIC ;
```

Explanation:

- RETIREMENT_BENEFITS is a function (method) that takes a given EMP instance as its argument and produces a result of type NUMERIC. *Note:* The parentheses are required; they surround a commalist (empty in the example) of data type specifications for additional arguments to the function.
- The code that implements the method is written in C plus embedded SQL/X and resides in an operating system file (whose name can be specified explicitly as part of the CREATE TABLE statement; if it is not, a default name derived from the method name is assumed).
- The method is invoked by a CALL statement of the form

```
CALL method ( argument-commalist ) ON variable INTO variable ;
```

For example:

```
CALL RETIREMENT_BENEFITS ( ) ON E INTO F ;
```

Here E and F are program variables; E contains the OID of some particular EMP instance, and the OID of the result of executing the method is assigned to F.

- Note, incidentally, that RETIREMENT_BENEFITS is *not* a "derived column." SQL/X does not currently support derived columns (though such support is promised for a future release).

The final extension is to permit the definition of **subclasses**. For example (refer to Fig. 25.3):

```
CREATE TABLE PERSON
     ( SS#         CHAR(9),
       BIRTHDATE   DATE,
       ADDRESS     CHAR(50) ) ;

CREATE TABLE EMP
    AS SUBCLASS OF PERSON
     ( EMP#        CHAR(5),
       ENAME       CHAR(20),
       SAL         NUMERIC,
       HOBBIES     SET OF ( ACTIVITY ),
       WORKS_FOR   COMPANY )
METHOD RETIREMENT_BENEFITS ( ) : NUMERIC ;
```

Explanation:

- EMP now has three additional columns (SS#. BIRTHDATE, ADDRESS) inherited from PERSON. If PERSON had any methods, it would inherit those too.

  Along with the data definition extensions sketched above, numerous data manipulation extensions are required also. They include:

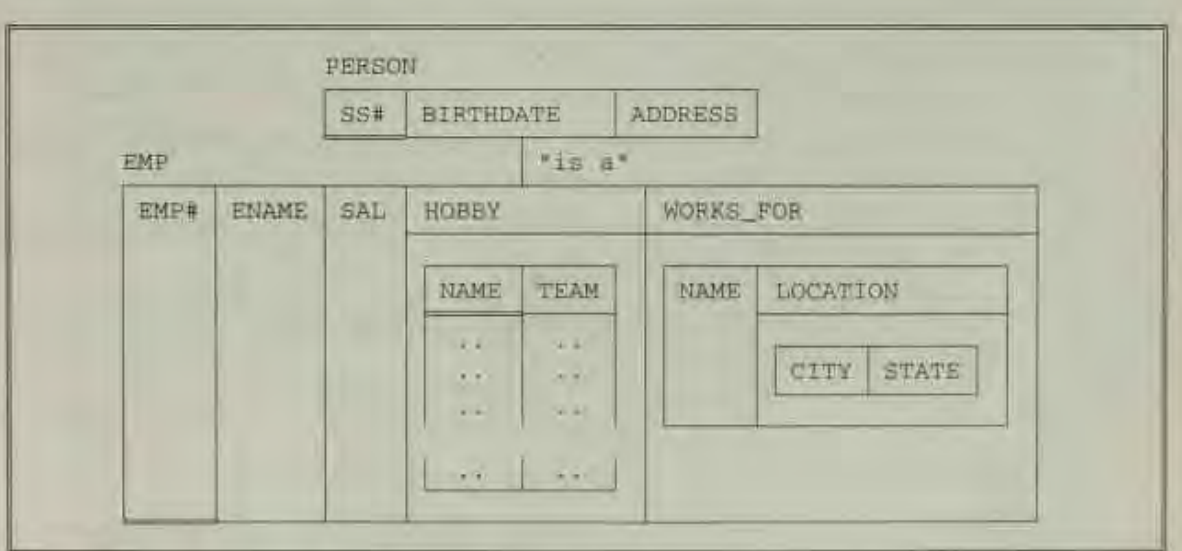

FIG. 25.3 SQL/X superclasses and subclasses (example)

- Path expressions (e.g., EMP.WORKS_FOR.LOCATION.STATE)—a new kind of scalar expression. For example:

```
SELECT ENAME, WORKS_FOR.NAME
FROM   EMP
WHERE  WORKS_FOR.LOCATION.STATE = 'CA' ;
```

  We remark that these path expressions go *down* the containment hierarchy, whereas the path expressions discussed briefly at the end of Chapter 23 go *up*.

- An extended kind of path expression that is multi-valued instead of single-valued (e.g., EMP.HOBBIES.NAME). Example:

```
SELECT ENAME, WORKS_FOR.NAME
FROM   EMP
WHERE  { 'Soccer', 'Bridge' } SUBSETEQ HOBBIES.NAME ;
```

  The operator SUBSETEQ means "proper subset of or equal to" (see below).

- The ability to write literal instances of complex objects. One simple example appears in the WHERE clause just shown. Here is a more complex example (a literal EMP):

```
( 'E001', 'Smith', $50000,
          { ( 'Soccer', 11 ), ( 'Bridge', 4 ) },
                              ( 'IBM', ( 'San Jose', 'CA' ) ) )
```

- Table comparison operators such as SUBSET, SUBSETEQ, SETEQ, etc. (see example above).

- Operations for traversing the class hierarchy. For example:

```
SELECT SS#, ENAME
FROM   EMP ;
```

  The FROM clause "FROM *T*" allows the user to refer to all columns of *T*, including those inherited from superclasses (at any level) of *T*. By contrast, the FROM clause "FROM ALL *T*" allows the user to refer to columns of subclasses (at any level) of *T* as well. For example:

```
SELECT SS#, BIRTHDATE, ENAME
FROM   ALL PERSON ;
```

  Note: It is not clear exactly what kind of object results from evaluating the expression "ALL PERSON"—perhaps a set of tables?

- The ability to invoke methods within WHERE clauses etc., in order to condition access (not supported at the time of writing).

- The ability to access (retrieve, update) individual values within multi-valued column values. For example:

```
UPDATE EMP
SET    HOBBIES = HOBBIES - { 'Soccer,' 11 }
WHERE  ENAME = 'Smith' ;
```

  (*cf.* the discussion of relational assignment and relational update operations in Chapter 6).

We now proceed to examine some consequences of the foregoing. **Please understand clearly that the following remarks are NOT intended as an attack on SQL/X *per se;*** rather, they are applicable to any system or language that attempts to equate relations and object classes. We use SQL/X merely by way of illustration.

1. The first point is that this system is a long way from being a true OO system. (As an aside, we remark that it is not really a "nested relation" or "RVA" system either—see Chapter 19—because of the points discussed in paragraphs 4 and 5 below.) It might best be characterized as **an SQL system with extensions**—which means, first and foremost, that it *is* an SQL system. *Note:* This comment is not meant as a criticism; it is just that an appreciation of this point will help the reader to understand the points that follow.
2. SQL/X "objects" are *rows in tables* (and are thus totally unencapsulated); the corresponding "instance variables" are *columns. Note:* Again, these comments are not necessarily criticisms—they are just observations.
3. Note, therefore, that whereas a pure OO class has methods and no (public) instance variables, an SQL/X "class" has public instance variables and does not necessarily have any methods. Thus, the "relation = class" equation looks a little suspect already.
4. We remark that there is a difference in kind between the column definitions (e.g)

   ```
   ENAME CHAR(20)
   ```

   and

   ```
   WORKS_FOR COMPANY
   ```

   "CHAR(20)" is a true data type (equivalently, a true— albeit primitive—domain); it places a time-independent constraint on the values that can appear in column ENAME. "COMPANY" by contrast is *not* a true data type; the constraint it places on the values that can appear in column WORKS_FOR is *time-dependent* (it depends, obviously, on the set of values currently appearing in table COMPANY). In fact, the "class *vs.* instance *vs.* collection" distinctions are muddied here, and the concept of true domains has been lost. The full implications of this state of affairs are unclear.
5. SQL/X objects are regarded (in general) as containing other objects—i.e., they are containment hierarchies. In fact, however, subobject sharing is done via pointers (OIDs), and users must understand this point clearly. For instance, referring back to the example of EMP objects "containing" COMPANY objects: Suppose the user updates one particular EMP object, changing (say) the name of the "contained" COMPANY object. Then that change will immediately be propagated to all other EMP objects for employees of the same company.

   Here are some further implications and questions arising from this same point:

   - Can we insert a new EMP instance and specify a value for the contained COMPANY instance that does not currently exist in the COMPANY table? If

the answer is yes, the definition of column WORKS_FOR as being of type COMPANY does not mean very much, since it does not significantly constrain the INSERT operation. If the answer is no, the INSERT operation becomes unnecessarily complex—the user has to specify, not just a legal COMPANY.NAME "foreign key" value (as would be required in the analogous relational situation), but an entire legal COMPANY object. Moreover, specifying an entire COMPANY object means—at best—having to tell the system something it already knows; at worst, it means that if the user makes a mistake, the INSERT will fail when it could perfectly well have succeeded.

- Suppose we want a "DELETE RESTRICTED" rule for COMPANY (i.e., an attempt to delete a company must fail if the company has any employees). Presumably this rule must be enforced by a suitable method, *M* say. Furthermore, native SQL DELETE operations must not be performed on COMPANY objects other than within that method *M*. How is this latter requirement enforced?
- Analogous remarks apply to "DELETE CASCADES," though possibly not to "DELETE NULLIFIES."
- DELETE on an EMP presumably does *not* delete the corresponding COMPANY, despite the pretense that the corresponding COMPANY is contained within the EMP.
- It follows from all of the above that we are not really talking about the relational model any more. The fundamental data object is not a relation containing values, it is a table containing values *and pointers*. As a consequence, operators such as projection and join require careful and subtle reinterpretation. This is a big issue, and we therefore make it our next major point.

6. Let us simplify the EMP table slightly, as follows (the notation is intended to be self-explanatory):

```
EMP ( EMP#, SAL, WORKS_FOR ( NAME, LOCATION ( CITY, STATE ) ) )
```

Now suppose we project this EMP table over SAL and the corresponding company CITY. Does the result look like this?—

```
( SAL, ( ( CITY ) ) )
```

or like this?—

```
( SAL, CITY )
```

Refer to Fig. 25.4.

- If it is the former (which is more logically correct)—see the left-hand side of the figure—then what is the name of the column that contains the relation that contains the CITY value? And what is the name of the column that contains the relation that contains the relation that contains the CITY value? *Note:* Lest the reader be tempted to reply "LOCATION.CITY" and "WORKS_FOR.LOCATION.CITY," respectively, we point out that these con-

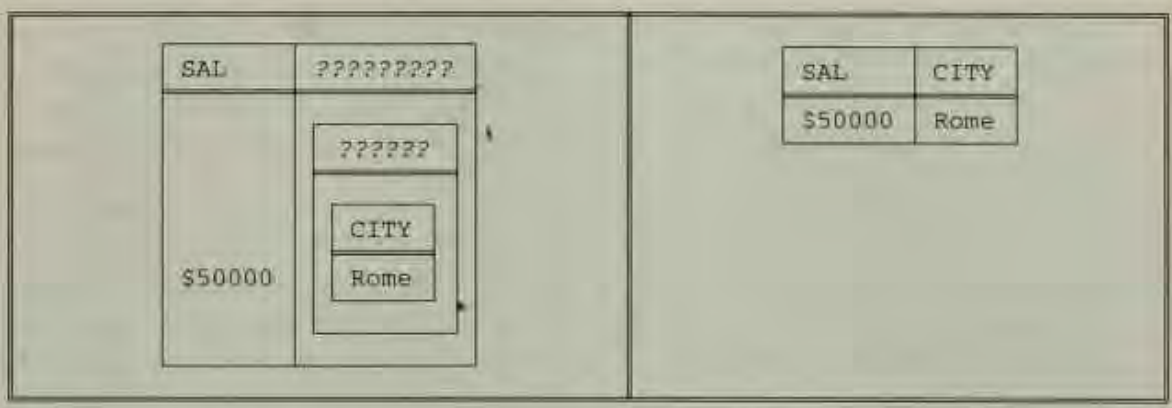

FIG. 25.4 Which projection is correct?

structs are not proper column names. They cannot be used in CREATE TABLE, after all.

- If it is the latter—see the right-hand side of the figure—then do we conclude that hierarchic objects make sense only at the "base object" level and that, as soon as we create any derived objects, we are back in the relational world? If so, why did we leave it in the first place?
- Assuming that the left-hand result, with heading (SAL,((CITY))), is indeed the correct one, can we now join that result to the CITYSTATE table over cities? That is, is the following comparison legal?

```
CITY = ( ( CITY ) )
```

Note that the CITY values on the right-hand side will at least have to be doubly unnested before the actual comparison can be done (for otherwise the comparison will certainly either give *false* or else raise a type error).

The foregoing discussion illustrates the two points that (a) if join is going to apply to hierarchic objects, it is going to require very careful definition, and (b) even if that definition can be properly worked out, the resultant operation is likely to be quite complex. In particular, the rules regarding the kind of object that is produced by the join are likely to be *very* complicated. The details are left to the reader to meditate on.

7. Referring again to the projection example from the previous discussion: Whichever is the correct result, **what class is it? What methods apply?**

In our example, EMPs had only one method, namely RETIREMENT_BENEFITS, and that method clearly does not apply to the result of the projection; in fact, it hardly seems reasonable that *any* methods that applied to EMP objects would apply. And there certainly are no others. So it looks as if we are forced to conclude that *no methods at all* apply to the result of a projection—i.e., the result, whatever it is, is not really an object class at all. (We might *say* it is an object class, but that does not

make it one!—it will have instance variables and no methods, whereas we have already observed that a true object class has methods and no instance variables.)

In fact, it is quite clear that when people equate relations and object classes, it is specifically *base* relations they are referring to.* The relational **closure** property applies to relational tables but not to object classes.

8. Following on from the previous point: Suppose we are given a base relation *R* and we project it over *all* of its columns. In the version of the relational algebra defined in Chapter 6 of this book, the syntax for such a projection is simply *R*. So the expression "*R*" is now ambiguous! If we think of it as referring to base relation *R*, then it is an object class, with methods. If we think of it as referring to the projection of base relation *R* over all of its columns, then it is not an object class and it has no methods. How do we tell the difference?
9. Finally, it has to be said that SQL/X represents a *major* increase in complexity *vis-ă-vis* plain SQL. Such additional complexity is an inevitable consequence of making the wrong equation (i.e., "relation = object class" instead of "domain = object class").

25.5 Benefits of True Rapprochement

In this section we briefly summarize the benefits of a true OO/relational rapprochement (i.e., a rapprochement that exploits the relational concept of domains properly, not one that makes the mistaken equation of relations and object classes).

First of all, the system is still a relational system, so all the usual relational benefits apply [3.2]. In particular, the advantage of *familiarity* applies—the OO capabilities represent a very natural "extension" (actually not an extension at all) to the relational model.

Second, the system is also an OO system, which implies the following additional benefits:

- *Open architecture (extendability):* Users are no longer limited to the predefined, builtin data types. The system can be tailored and extended in arbitrary ways to meet the needs of specific application areas.
- *User-defined data types and functions, with inheritance:* These in a nutshell are **the** advantages of OO systems in general.
- *Strong typing:* Proper domain support provides a sound basis for catching type violations (preferably at compilation time), as discussed in Chapter 19.
- *Improved performance:* Performance is improved because methods are executed

* Or at least *named* relations. SQL/X does support *views*, which are regarded as "virtual classes"—see the annotation to reference [25.11]. Virtual classes do not inherit methods from the classes from which they are derived, however.

by the server, not the client.* As a trivial example, consider the query "Find all books with more than 20 chapters." In a traditional relational system, books might be represented as "BLOBs" (a **BLOB** or *basic large object* is essentially just an arbitrarily long byte string), and the client will have to retrieve each book in turn and scan it to see if it has more than 20 chapters. But with proper OO support, the "number of chapters" method will be executed by the server, and only those books actually desired will be transmitted to the client.

Note: The foregoing is not really an argument for OO, it is an argument for *stored procedures* (see Chapter 21). A traditional relational system with stored procedures will give the same performance benefits here as an OO system with methods.

- *Improved productivity:* Once the appropriate object classes (domains) have been defined, users of those classes can obviously be much more productive, both in application development and—probably more important—in application maintenance also.
- *Object-level recovery, concurrency, and security:* As pointed out in Chapter 22, objects—that is, individual objects, not object classes—are the natural unit for recovery, concurrency, and security purposes in an OO system (integrity constraints permitting).
- *Accessibility:* Complex objects now become available via the relational query language (probably SQL) to end users.

Furthermore, all of the criticisms we leveled at OO systems in previous chapters no longer apply. To be specific, the system can now support all of the following without any undue difficulty:

- *Ad hoc* query
- Dual-mode access (i.e., using the same language for programmed and interactive database access)
- Relationships of degree greater than two
- Methods that span classes
- Symmetric exploitation
- Declarative integrity constraints
- Integrity constraints that span classes
- "Foreign key rules" (DELETE CASCADES, etc.)
- Semantic optimization

In addition:

- OIDs and pointer chasing are now totally "under the covers" and hidden from the user
- "Difficult" OO questions (e.g., what does it mean to join two objects?) go away

---

* Not true at the time of writing for SQL/X. A future release will offer an option by which the DBA can specify where a given method is to be executed.

- The benefits of encapsulation still apply (to values within relations, not to relations themselves)
- Relational systems can now handle "complex" application areas such as CAD/CAM, as previously discussed

And the approach is conceptually clean.

25.6 Summary

The message of this chapter (and indeed its three predecessors) can be summed up as follows. First of all, OO unquestionably has some good ideas, namely:

- **User-defined data types** (including user-defined functions)
- **Inheritance** (behavioral, not structural)

However, it also has some ideas that would be better rejected, or at least kept under the covers:

- **OIDs** and pointer chasing
- Strong **procedural** (3GL) orientation
- Function **bundling**
- Complex and asymmetric treatment of **relationships**
- Lack of declarative **integrity** support

What we need to do is enhance relational systems to incorporate the good ideas of OO (but not the bad ones!), and we have argued that **domains** are the key to achieving this goal. Specifically, we have argued that the equation "domain = object class" is correct, and the equation "relation = object class" is incorrect.

References and Bibliography

See also the references in Chapter 19 and Chapters 22–24.

25.1 Malcolm Atkinson *et al.* "The Object-Oriented Database System Manifesto." Proc. First International Conference on Deductive and Object-Oriented Databases, Kyoto, Japan (1989). New York, N.Y.: Elsevier Science (1990).

One of the first attempts to build a consensus on what "the OO model" should include. See reference [25.10] for another attempt.

25.2 François Bancilhon. "A Logic-Programming/Object-Oriented Cocktail." *ACM SIGMOD Record 15*, No. 3 (September 1986).

To quote the introduction: "The object-oriented approach . . . seems to be particularly well fitted to [handling] new types of applications such as CAD, software [engineering], and AI. However, the natural extension to relational database technology is . . . the logic programming paradigm, [not] the object-oriented one. [This paper addresses the question of] whether the two paradigms are compatible."

25.3 David Beech. "A Foundation for Evolution from Relational to Object Databases." In Schmidt, Ceri, and Missikoff (eds.): *Extending Database Technology*. New York, N.Y.: Springer-Verlag (1988).

This is one of several papers that discuss the possibility of extending SQL to become some kind of "Object SQL" or "OSQL" (see also reference [25.7], which gives more detail on this particular proposal). In common with most such papers it has little or nothing to say regarding domains, closure, or relational operators such as join; i.e., it falls into the "relation = class" trap. It also makes a number of somewhat contentious claims regarding (among other things) the "smooth adaptation" of SQL. Here, for example, is an OSQL CREATE FUNCTION statement (taken from reference [25.7]):

```
CREATE FUNCTION RaiseAllSalaries (Integer incr) AS
       UPDATE Salary(e) = newsal
       FOR EACH Employee e, Integer newsal
       WHERE newsal = Salary(e) + incr ;
```

The following might seem to many people to be more in keeping with the style of conventional SQL:

```
CREATE FUNCTION RaiseAllSalaries (Integer incr) AS
       UPDATE Employee
       SET Salary = Salary + incr ;
```

As in SQL/X, "types" (tables) are not encapsulated. *Note:* One interesting aspect of OSQL is that (in contrast to some other OO languages) methods—i.e., functions—are themselves regarded as objects.

25.4 R. G. G. Cattell. "What Are Next-Generation DB Systems?" *CACM 34*, No. 10 (October 1991).

25.5 Nathan Goodman. "Object Oriented Database Systems." *InfoDB 4*, No. 3 (Fall 1989).

The previous edition of this book included the following quote from this paper:

"At this stage it is futile to compare [the] relational and object-oriented [approaches]. We have to compare like to like: apples to apples, dreams to dreams, theory to theory, and mature products to mature products."

And it (the previous edition) went on to say:

"Dreams aside, the relational approach has been in existence for some time: It rests on a very solid theoretical foundation, and it is the basis for a large number of mature products. The OO approach, by contrast, is new (at least in the database arena); it does not possess a theoretical foundation that can meaningfully be compared with the relational model, and the few products that exist can scarcely be described as mature. Thus, a great deal remains to be done before we can seriously begin to consider the question of whether OO technology will ever represent a viable alternative to the relational approach."

While many of the foregoing remarks are still applicable, matters have crystallized somewhat since the previous edition was published. In many ways, in fact, relational *vs.* OO can now be seen to be somewhat of an apples-and-oranges comparison; as explained in Chapter 24, a relational DBMS comes *ready for use*, whereas an OO DBMS by contrast is best thought of as a kind of *construction kit* for building application-specific DBMSs.

25.6 Michael Kifer, Won Kim, and Yehoshua Sagiv. "Querying Object-Oriented Databases." Proc. ACM International Conference on Management of Data, San Diego, Calif. (June 1992).

Proposes another "Object SQL" called XSQL.

25.7 Peter Lyngbaek *et al.* "OSQL: A Language for Object Databases." Technical Report HPL-DTD-91-4, Hewlett-Packard Company (January 15th, 1991).

See the annotation to reference [25.3].

25.8 Gail M. Shaw and Stanley B. Zdonik. "A Query Algebra for Object-Oriented Databases." Proc. 6th International Conference on Data Engineering (February 1990). Also Technical Report TR CS-89-19, Dept. of Computer Science, Brown University, Providence, RI (March 1989).

This paper serves to support the present writer's contention that any "OO algebra" will be inherently complex (because objects are complex). In particular, *equality* of arbitrarily nested hierarchic objects requires very careful treatment. The basic idea behind the specific proposals of this paper is that each query algebra operator produces a *relation*, in which each tuple contains OIDs for certain database objects. In the case of join, for example, each tuple contains OIDs of objects that match one another under the joining condition. Those tuples do not inherit any methods from the component objects.

25.9 David W. Shipman. "The Functional Data Model and the Data Language DAPLEX." *ACM TODS 6*, No. 1 (March 1981). Republished in M. Stonebraker (ed.): *Readings in Database Systems*. San Mateo, Calif.: Morgan Kaufmann (1988).

There have been several attempts to construct systems based on *functions* instead of relations, and DAPLEX is one of the best known. The functional approaches share certain ideas with the OO approach, including in particular a somewhat navigational (i.e., path-following) style of addressing objects that are functionally related to other objects (that are functionally related to other objects, etc.). Note, however, that a function in one of the so-called "functional" models is typically not a mathematical function at all; it might, for example, be multi-valued. In fact, considerable violence has to be done to the function concept in order to make it capable of all the things required of it in the "functional data model" context. Be that as it may, it seems to this writer that the functional approach and the OO approach are in large part the same thing under different names.

25.10 Michael Stonebraker *et al.* "Third Generation Database System Manifesto." *ACM SIGMOD Record 19*, No. 3 (September 1990).

In part, this paper is a response to—i.e., counterproposal to—the proposals of reference [25.1]. A quote: "Second generation systems made a major contribution in two areas, nonprocedural data access and data independence, and these advances must not be compromised by third generation systems." The following features are stated to be essential requirements of a "third generation DBMS" (the list is somewhat paraphrased):

1. Provide traditional DB services plus richer object structures and rules
   - Rich type system
   - Inheritance
   - Functions and encapsulation
   - Optional system-assigned tuple IDs [there seems to be some confusion here between object IDs and tuple IDs]
   - Rules (e.g., integrity rules), *not* tied to specific objects
2. Subsume second generation DBMSs
   - Navigation only as a last resort
   - Intensional and extensional set definitions [meaning collections that are maintained automatically by the system and collections that are maintained manually by the user]

- Updatable views
- Clustering, indexes, etc., hidden from the user

3. Support open systems
   - Multiple languages
   - Persistence
   - SQL [characterized as "intergalactic dataspeak" (!)]
   - Queries and results must be the lowest level of client/server communication

Regarding SQL support for "complex objects," incidentally, it seems to this writer that such support can be made part of a clean, abstract OO model only if the object design discipline from Section 24.3 is made an essential part of that model too. Assuming that such query capabilities *are* incorporated into the model, we could then extend the requirement to permit those capabilities to be invoked from within application code, thereby:

1. Moving away from the present 3GL orientation of OO systems, and
2. Providing support for the dual-mode principle, as in relational systems.

We would also have a basis for stating integrity constraints declaratively and providing declarative view support.

25.11 UniSQL Inc. *UniSQL/X Database Management System User's Manual: Release 1.2* (1991).

See Section 25.4 for an extended discussion of this product. It is worth noting that, precisely because (as stated in that section) it is really an extended SQL system, not a true OO DBMS, some of the criticisms leveled at OO systems in general in Chapter 24 and elsewhere do not apply to UniSQL in particular. To be more specific:

- UniSQL does support views ("virtual classes"). They have their own methods, however—i.e., they do not inherit methods from the tables (classes) from which they are derived. Incidentally, the manual talks in terms of "classes and virtual classes," which suggests that virtual classes are not classes. This situation is reminiscent of the SQL standard's talk of "tables and views" (see Chapter 3), which suggests that views are not tables.
- In fact, UniSQL does support derived objects in general—i.e., the relational operators join, union, etc., are all supported. However, those derived objects are not regarded as classes, and it is not clear exactly what *kind* of objects (e.g., how nested they are) those operations produce.
- UniSQL does have a single language (SQL/X) for both programmed and interactive access to the database. Moreover, that language is set-level, and SQL/X requests are thus optimizable. It also supports symmetric exploitation.
- UniSQL does have support for missing information—it provides both SQL-style nulls and (limited) support for user-defined default values.
- It also presumably supports a predefined system catalog.
- Finally, it provides a separate CREATE CLUSTER statement, which means that (unlike some OO systems) it does not make the mistake of undermining data independence by letting the logical database definition dictate what physical clustering to use.

25.12 Carlo Zaniolo. "The Database Language GEM." Proc. 1983 ACM SIGMOD International Conference on Management of Data, San Jose, Calif. (May 1983). Republished in

M. Stonebraker (ed.): *Readings in Database Systems*. San Mateo, Calif.: Morgan Kaufmann (1988).

GEM is an acronym, standing for "General Entity Manipulator." It is effectively an extension to QUEL that supports non1NF relations (i.e., relations with set-valued attributes) and relations with alternative attributes (e.g., exempt EMPs have a SALARY attribute, nonexempt EMPs have HOURLY_WAGE and OVERTIME attributes). It also makes use of the OO idea that objects conceptually contain other objects (instead of foreign keys that reference those other objects), and extends the familiar dot notation to provide a simple way of referring to attributes of such contained objects (in effect, by implicitly traversing certain preferred join paths). For example, the qualified name EMP.DEPT.BUDGET could be used to refer to the budget of the department of some given employee. Many other systems, including POSTGRES [25.13-25.16] and UniSQL [25.11], have adopted and/or adapted this idea.

Extendable Systems

A variety of research projects have been under way for some years to build systems that are essentially relational and yet (like OO systems) are *extendable*—that is, they allow knowledgeable users (sometimes called "database customizers") to tailor the system in various ways, e.g., by defining their own data types, their own functions, their own storage structures, and so on. Two of the best known projects are **POSTGRES** from the University of California at Berkeley and **Starburst** from IBM Research. The following is a brief selection from the numerous publications on these two projects. Note, however, that neither of these systems adheres to the "domain = object class" equivalence advocated in the body of this chapter.

25.13 Michael Stonebraker and Lawrence A. Rowe. "The Design of POSTGRES." Proc. ACM SIGMOD International Conference on Management of Data, Washington, D.C. (June 1986).

The stated objectives of POSTGRES are:

1. To provide better support for complex objects
2. To provide user extendability for data types, operators, and access methods
3. To provide active database facilities (alerters and triggers) and inferencing support
4. To simplify the DBMS code for crash recovery
5. To produce a design that can take advantage of optical disks, multiple-processor workstations, and custom-designed VLSI chips
6. To make as few changes as possible (preferably none) to the relational model

25.14 Michael Stonebraker. "The Design of the POSTGRES Storage System." Proc. 13th International Conference on Very Large Data Bases, Brighton, England (September 1987).

25.15 Lawrence A. Rowe and Michael R. Stonebraker. "The POSTGRES Data Model." Proc. 13th International Conference on Very Large Data Bases, Brighton, England (September 1987).

25.16 Michael Stonebraker and Greg Kemnitz. "The POSTGRES Next Generation Database Management System." *CACM 34*, No. 10 (October 1991).

25.17 Bruce Lindsay, John McPherson, and Hamid Pirahesh. "A Data Management Extension Architecture." Proc. ACM SIGMOD International Conference on Management of Data, San Francisco, Calif. (May 1987).

Describes the overall architecture of the Starburst prototype. Starburst "facilitates the

implementation of data management extensions for relational database systems." Two kinds of extensions are described in this paper, user-defined storage structures and access methods, and user-defined integrity constraints and triggered procedures. However (to quote the paper), "there are, of course, other directions in which it is important to be able to extend [DBMSs, including] user-defined abstract data types [and] query evaluation techniques." See reference [25.18] below.

25.18 Laura M. Haas, J. C. Freytag, G. M. Lohman, and Hamid Pirahesh. "Extensible Query Processing in Starburst." Proc. ACM SIGMOD International Conference on Management of Data, Portland, Ore. (June 1989).

The aims of the Starburst project expanded somewhat after reference [25.17] was first written: "Starburst provides support for adding new storage methods for tables, new types of access methods and integrity constraints, new data types, functions, and new operations on tables." The system is divided into two major components, Core and Corona, corresponding respectively to the RSS and RDS in the original System R prototype [8.12–8.16]. Core supports the extendability functions described in reference [25.17]. Corona supports the Starburst query language Hydrogen, which is a dialect of SQL that (a) eliminates most of the implementation restrictions of System R SQL, (b) is much more orthogonal than System R SQL, (c) supports recursive queries, and (d) is user-extendable. The paper includes an interesting discussion of "query rewrite," i.e., Hydrogen transformation rules that are used by the optimizer to convert queries to a semantically equivalent but more efficient form (see Chapter 18). See also reference [18.44].

25.19 Guy M. Lohman *et al.* "Extensions to Starburst: Objects, Types, Functions, and Rules." *CACM 34*, No. 10 (October 1991).

APPENDIXES

The following appendixes cover a somewhat mixed bag of topics. Appendix A provides a tutorial survey of common storage structures and access techniques. Appendix B gives a brief overview of a commercial relational implementation—namely, IBM's DB2 product—in order to give some idea of how a real relational DBMS actually works. Appendix C is an introduction to the ideas underlying logic-based systems (i.e., systems that attempt to marry ideas from the logic programming world with database ideas). Finally, Appendix D presents a list of the more important abbreviations and acronyms introduced in the text, together with their meanings.

APPENDIX

A Storage Structures and Access Methods

A.1 Introduction

In this appendix we present a tutorial survey of techniques for physically representing and accessing the database on the disk. (*Note:* We use the term "disk" throughout to stand generically for all direct-access media, including, e.g., RAID arrays, mass storage, and more recently optical disks, as well as conventional moving-head magnetic disks *per se*). The reader is assumed to have a basic familiarity with disk architecture and to understand what is meant by such terms as **seek time, rotational delay, cylinder, track, read/write head,** and so on. Good tutorials on such material can be found in many places; see, for example, reference [A.4].

The basic point motivating all storage structure and access method technology is that disk access times are *much* slower than main memory access times. Typical seek times and rotational delays are both on the order of 10 milliseconds or so, and typical data transfer rates are somewhere in the range 5 to 10 million bytes per second; main memory access is likely to be at least four or five orders of magnitude faster than disk access on any given system. An overriding performance objective is thus *to minimize the number of disk accesses* (disk I/O's). This appendix is concerned with techniques for achieving that objective—i.e., techniques for arranging stored data on the disk so that a required piece of data, say a required stored record, can be located in as few I/O's as possible.

As already suggested, any given arrangement of data on the disk is referred to as a **storage structure**. Many different storage structures can be and have been devised, and of course different structures have different performance characteristics; some are good for some applications, others are good for others. There is no single structure that is optimal for all applications. It follows that a good system should support a variety of different structures, so that different portions of the database can be stored in different ways, and the storage structure for a given portion can be changed as performance requirements change or become better understood.

The structure of the presentation is as follows. Following this introductory section, Section A.2 explains in outline what is involved in the overall process of locating and

accessing some particular stored record,* and identifies the major software components involved in that process. Section A.3 then goes into a little more detail on two of those components, the *file manager* and the *disk manager*. Those two sections (A.2 and A.3) need only be skimmed on a first reading, if the reader desires; a lot of the detail they contain is not really required for an understanding of the subsequent material. The next four sections (Sections A.4–A.7) should not be just skimmed, however, since they represent the most important part of the entire discussion; they describe some of the most commonly occurring storage structures found in present-day systems, under the headings *indexing*, *hashing*, *pointer chains*, and *compression techniques*, respectively. Finally, Section A.8 presents a summary and a brief conclusion.

Note: The emphasis throughout is on *concepts*, not detail. The objective is to explain the general idea behind such notions as indexing, hashing, etc., without getting too bogged down in the details of any one specific system or technique. The reader who needs such detail is directed to the books and papers listed in the References and Bibliography section at the end of the appendix.

A.2 Database Access: An Overview

Before we get into our discussion of storage structures *per se*, we first briefly consider what is involved in the overall process of database access in general. Locating a specific piece of data in the database and presenting it to the user involves several layers of data access software. Of course, the details of those layers vary considerably from system to system, and so too does the terminology, but the principles are fairly standard and can be explained in outline as follows (refer to Fig. A.1).

1. First, the DBMS decides what stored record is required, and asks the **file manager** to retrieve that record. (We assume for the purposes of this simple explanation that the DBMS is able to pinpoint the exact record desired ahead of time. In practice it might need to retrieve a set of several records and search through those records in main memory to find the specific one desired. In principle, however, this only means that the sequence of steps 1–3 must be repeated for each stored record in that set.)
2. The file manager in turn decides what page contains the desired record, and asks the **disk manager** to retrieve that page.
3. Finally, the disk manager determines the physical location of the desired page on the disk, and issues the necessary disk I/O request. *Note:* Sometimes, of course, the required page will already be in a buffer in main memory as the result of a previous retrieval, in which case it obviously should not be necessary to retrieve it again.

Loosely speaking, therefore, the DBMS has a view of the database as a collection of stored records, and that view is supported by the file manager; the file manager, in

* Throughout this appendix, since our discussions are all at the storage level, we use storage-level terminology (files, records, fields, etc.) in place of their relational analogs.

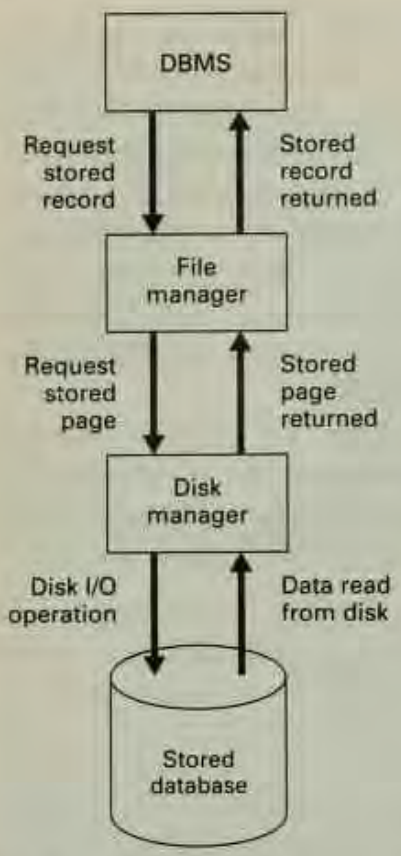

FIG. A.1 The DBMS, file manager, and disk manager

turn, has a view of the database as a collection of pages, and that view is supported by the disk manager; and the disk manager has a view of the disk "as it really is." The three subsections below amplify these ideas somewhat. Section A.3 then goes into more detail on the same topics.

Disk Manager

The disk manager is a component of the underlying operating system. It is the component responsible for all physical I/O operations (in some systems it is referred to as the "basic I/O services" component). As such, it clearly needs to be aware of **physical disk addresses**. For example, when the file manager asks to retrieve some specific page *p*, the disk manager needs to know exactly where page *p* is on the physical disk. However, the user of the disk manager—namely, the file manager—does *not* need to know that information. Instead, the file manager regards the disk simply as a logical collection of **page sets,** each one consisting of a collection of fixed-size pages. Each page set is identified by a unique **page set ID**. Each page, in turn, is identified by a **page number** that is unique within the disk; distinct page sets are disjoint (i.e., do not have any pages in common). The mapping between page numbers and physical disk addresses is understood and maintained by the disk manager. The major advantage of this arrangement (not the only one) is that all device-specific code can be isolated within a single system

component, namely the disk manager, and all higher-level components—in particular, the file manager—can thus be *device-independent.*

As just explained, the complete set of pages on the disk is divided into a collection of disjoint subsets called page sets. One of those page sets, the **free space** page set, serves as a pool of available (i.e., currently unused) pages; the others are all considered to contain significant data. The allocation and deallocation of pages to and from page sets is performed by the disk manager on demand from the file manager. The operations supported by the disk manager on page sets—i.e., the operations the file manager is able to issue—include the following:

- Retrieve page *p* from page set *s*
- Replace page *p* within page set *s*
- Add a new page to page set *s* (i.e., acquire an empty page from the free space page set and return the new page number *p*)
- Remove page *p* from page set *s* (i.e., return page *p* to the free space page set)

The first two of these operations are of course the basic page-level I/O operations the file manager needs. The other two allow page sets to grow and shrink as necessary.

File Manager

The file manager uses the disk manager facilities just described in such a way as to permit its user—*viz.*, the DBMS—to regard the disk as a collection of **stored files** (remember from Chapter 1 that a stored file is the collection of all occurrences of one type of stored record). Each page set will contain one or more stored files. *Note:* The DBMS does need to be aware of the existence of page sets, even though it is not responsible for managing them in detail, for reasons indicated in the next subsection. In particular, the DBMS needs to know when two stored files share the same page set or when two stored records share the same page.

Each stored file is identified by a **file name** or **file ID,** unique at least within its containing page set, and each stored record, in turn, is identified by a **record number** or **record ID** (RID), unique at least within its containing stored file. (In practice, record IDs are usually unique, not just within their containing file, but actually within the entire disk, since they typically consist of the combination of a page number and some value that is unique within that page. See Section A.3, later.)

The operations supported by the file manager on stored files—i.e., the operations the DBMS is able to issue—include the following:

- Retrieve stored record *r* from stored file *f*
- Replace stored record *r* within stored file *f*
- Add a new stored record to stored file *f* and return the new record ID *r*
- Remove stored record *r* from stored file *f*
- Create a new stored file *f*
- Destroy stored file *f*

Using these primitive file management operations, the DBMS is able to build and manipulate the storage structures that are the principal concern of this appendix (see Sections A.4–A.7).

Note: In some systems the file manager is a component of the underlying operating system, in others it is packaged with the DBMS. For our purposes the distinction is not important. However, we remark in passing that, although operating systems do invariably provide such a component, it is often the case that the general-purpose file manager provided by the operating system is not ideally suited to the requirements of the special-purpose "application" that is the DBMS. For more discussion of this topic, see reference [A.44].

Clustering

We should not leave this overview discussion without a brief mention of the subject of *data clustering*. The basic idea behind clustering is to try and store records that are logically related (and therefore frequently used together) physically close together on the disk. Physical data clustering is an extremely important factor in performance, as can easily be seen from the following. Suppose the stored record most recently accessed is record *r1*, and suppose the next stored record required is record *r2*. Suppose also that *r1* is stored on page *p1* and *r2* is stored on page *p2*. Then:

1. If *p1* and *p2* are one and the same, then the access to *r2* will not require any physical I/O at all, because the desired page *p2* will already be in a buffer in main memory.
2. If *p1* and *p2* are distinct but physically close together—in particular, if they are physically adjacent—then the access to *r2* will require a physical I/O (unless, of course, *p2* also happens to be in a main memory buffer), but the seek time involved in that I/O will be small, because the read/write heads will already be close to the desired position. In particular, the seek time will be *zero* if *p1* and *p2* are in the same cylinder.

As an example of clustering, we consider the usual suppliers-and-parts database:*

- If sequential access to all suppliers in supplier number order is a frequent application requirement, then the supplier records should be clustered such that the supplier S1 record is physically close to the supplier S2 record, the supplier S2 record is physically close to the supplier S3 record, and so on. This is an example of **intra-file** clustering: The clustering is applied within a single stored file.
- If, on the other hand, access to some specific supplier together with all shipments for that supplier is a frequent application requirement, then supplier and shipment records should be stored interleaved, with the shipment records for supplier S1 physically close to the supplier S1 record, the shipment records for supplier S2

* We assume for simplicity throughout this appendix that each individual supplier, part, or shipment tuple maps to a single stored record on the disk (barring explicit statements to the contrary).

physically close to the supplier S2 record, and so on. This is an example of **inter-file** clustering: The clustering is applied across more than one stored file.

Of course, a given file or set of files can be physically clustered in one and only one way at any given time.

The DBMS can support clustering, both intra- and inter-file, by storing logically related records on the same page where possible and on adjacent pages where not (this is why the DBMS must know about pages as well as stored files). When the DBMS creates a new stored record, the file manager must allow it to specify that the new record be stored "near"—i.e., on the same page as, or at least on a page logically near to—some existing record. The disk manager, in turn, will do its best to ensure that two pages that are logically adjacent are physically adjacent on the disk. See Section A.3.

Of course, the DBMS can only know what clustering is required if the database administrator is able to tell it. A good DBMS should allow the DBA to specify different kinds of clustering for different files. It should also allow the clustering for a given file or set of files to be changed if the performance requirements change. Furthermore, of course, any such change in physical clustering should not require any concomitant changes in application programs, if data independence is to be achieved.

A.3 Page Sets and Files

As explained in the previous section, a major function of the disk manager is to allow the file manager to ignore all details of physical disk I/O and to think in terms of (logical) "page I/O" instead. This function of the disk manager is referred to as **page management**. We present a very simple example to show how page management is typically handled.

Consider the suppliers-and-parts database once again. Suppose that the desired logical ordering of records within each table is (loosely) *primary key sequence*—that is, suppliers are required to be in supplier number order, parts in part number order, and shipments in part number order within supplier number order.* To keep matters simple, suppose too that each stored file is stored in a page set of its own, and that each stored record requires an entire page of its own. Suppose also that the disk contains a total of 64K = 65,536 pages (not at all unreasonable, by the way). Now consider the following sequence of events.

1. Initially the database contains no data at all. There is only one page set, the free space page set, which contains all the pages on the disk—except for page zero, which is special (see later). The remaining pages are numbered sequentially from one.
2. The file manager requests the creation of a page set for supplier records, and inserts

* We say "loosely" because the term *primary key sequence* is not well-defined if the primary key is composite. In the case of shipments, for example, it might mean either part number order within supplier number order or the other way about.

the five supplier records for suppliers S1–S5. The disk manager removes pages 1–5 from the free space page set and labels them "the suppliers page set."

3. Similarly for parts and shipments. Now there are four page sets: the suppliers page set (pages 1–5), the parts page set (pages 6–11), the shipments page set (pages 12–23), and the free space page set (pages 24, 25, 26, . . .). The situation at this point is as shown in Fig. A.2.

To continue with the example:

4. Next, the file manager inserts a new supplier stored record (for a new supplier, supplier S6). The disk manager locates the first free page in the free space page set—namely, page 24—and adds it to the suppliers page set.
5. The file manager deletes the stored record for supplier S2. The disk manager returns the page for supplier S2 (page 2) to the free space page set.
6. The file manager inserts a new part stored record (for part P7). The disk manager locates the first free page in the free space page set—namely, page 2—and adds it to the parts page set.
7. The file manager deletes the stored record for supplier S4. The disk manager returns the page for supplier S4 (page 4) to the free space page set.

And so on. The situation at this juncture is as illustrated in Fig. A.3. The point about that figure is the following: After the system has been running for a while, it can no longer be guaranteed that pages that are logically adjacent are still physically adjacent, even if they started out that way. For this reason, the logical sequence of pages in a given page set must be represented, not by physical adjacency, but by *pointers*. Each page will contain a **page header,** i.e., a set of control information that includes (among

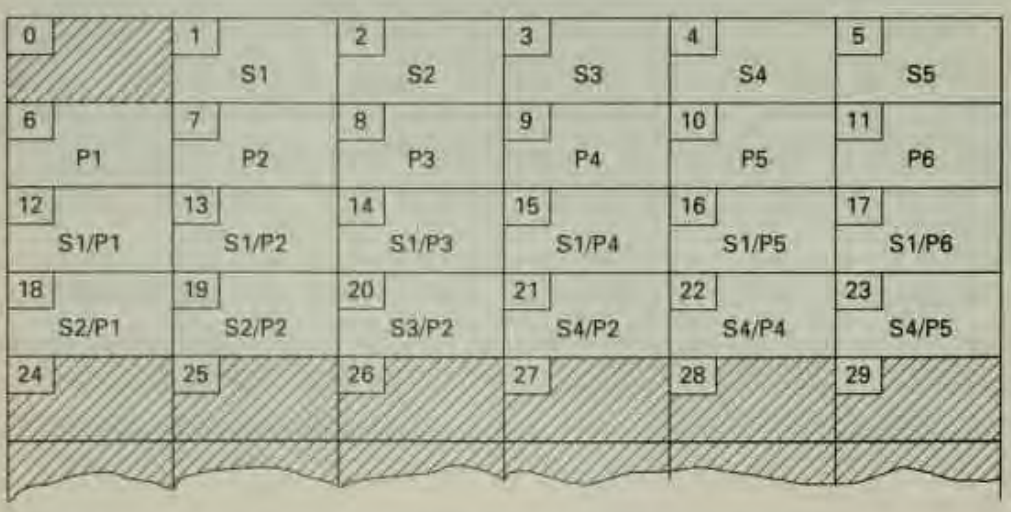

FIG. A.2 Disk layout after creation and initial loading of the suppliers-and-parts database

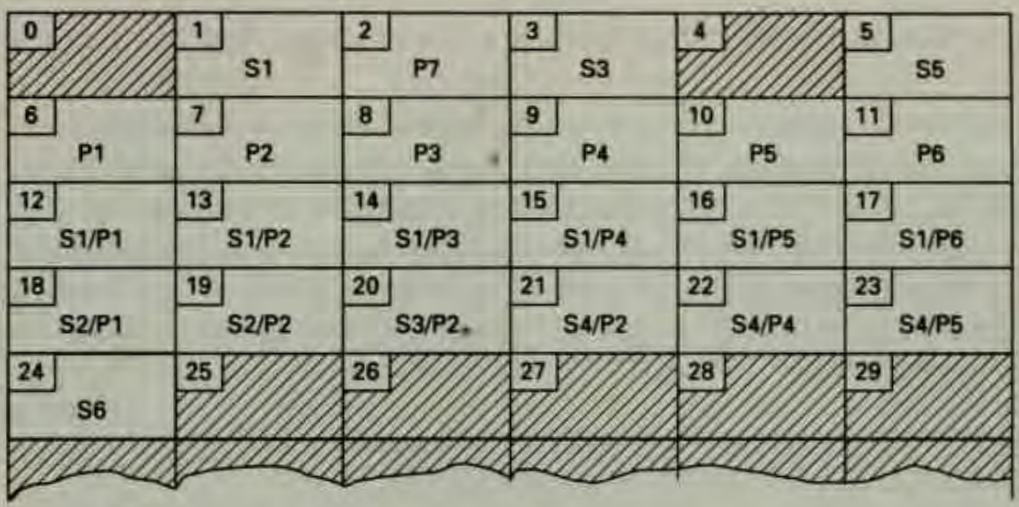

FIG. A.3 Disk layout after inserting supplier S6, deleting supplier S2, inserting part P7, and deleting supplier S4

other things) the physical disk address of the page that immediately follows that page in logical sequence. See Fig. A.4.

Points arising from the foregoing example:

- The page headers—in particular, the "next page" pointers—are managed by the disk manager; they should be completely invisible to the file manager.
- As explained in the subsection on clustering at the end of Section A.2, it is desirable to store logically adjacent pages in physically adjacent locations on the disk (as far as possible). For this reason, the disk manager normally allocates and deallocates pages to and from page sets, not one at a time as suggested in the ex-

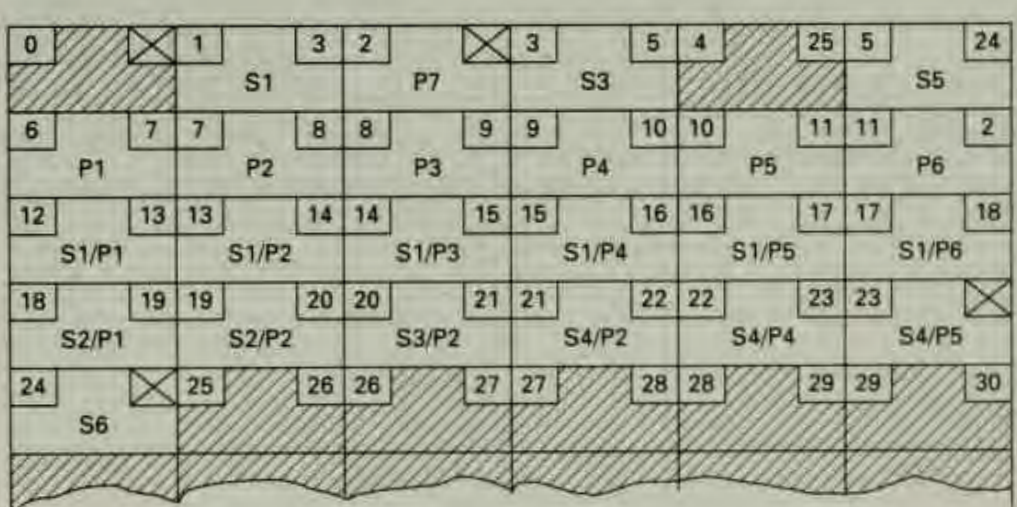

FIG. A.4 Fig. A.3 revised to show "next page" pointers (top right corner of each page)

ample, but rather in physically contiguous groups or **extents** of (say) 64 pages at a time.

- The question arises: How does the disk manager know where the various page sets are located?—or, more precisely, how does it know, for each page set, where the (logically) first page of that page set is located? (It is sufficient to locate the first page, of course, because the second and subsequent pages can then be located by following the pointers in the page headers.) The answer is that some fixed location on the disk—typically cylinder zero, track zero—is used to store a page that gives precisely that information. That page (variously referred to as the *disk table of contents*, the *disk directory*, the *page set directory*, or simply *page zero*) thus typically contains a list of the page sets currently in existence on the disk, together with a pointer to the first page of each such page set. See Fig. A.5.

Now we turn to the file manager. Just as the disk manager allows the file manager to ignore details of physical disk I/O and to think for the most part in terms of logical pages, so the file manager allows the DBMS to ignore details of page I/O and to think for the most part in terms of stored files and stored records. This function of the file manager is referred to as **stored record management**. We discuss that function very briefly here, once again taking the suppliers-and-parts database as the basis for our examples.

Suppose, then (rather more realistically now), that a single page can accommodate several stored records, instead of just one as in the page management example. Suppose too that the desired logical order for supplier records is supplier number order, as before. Consider the following sequence of events.

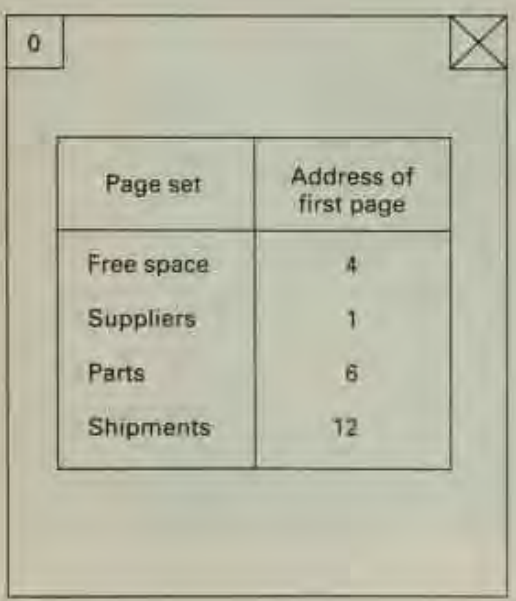

FIG. A.5 The disk directory ("page zero")

1. First, the five stored records for suppliers S1–S5 are inserted and are stored together on some page *p*, as shown in Fig. A.6. Note that page *p* still contains a considerable amount of free space.
2. Now suppose the DBMS inserts a new supplier stored record (for a new supplier, say supplier S9). The file manager stores this record on page *p* (because there is still space), immediately following the stored record for supplier S5.
3. Next, the DBMS deletes the stored record for supplier S2. The file manager erases the S2 record from page *p*, and shifts the records for suppliers S3, S4, S5, and S9 up to fill the gap.
4. Next, the DBMS inserts a new supplier stored record for another new supplier, supplier S7. Again the file manager stores this record on page *p* (because there is still space); it places the new record immediately following that for supplier S5, shifting the record for supplier S9 down to make room. The situation at this juncture is illustrated in Fig. A.7.

And so on. The point of the example is that the logical sequence of stored records within any given page can be represented by *physical* sequence within that page. The file manager will shift individual records up and down to achieve this effect, keeping all data records together at the top of the page and all free space together at the bottom. (The logical sequence of stored records *across* pages is of course represented by the sequence of those pages within their containing page set, as described in the page management example earlier.)

As explained in Section A.2, stored records are identified internally by *record ID* or RID. Fig. A.8 shows how RIDs are typically implemented. The RID for a stored

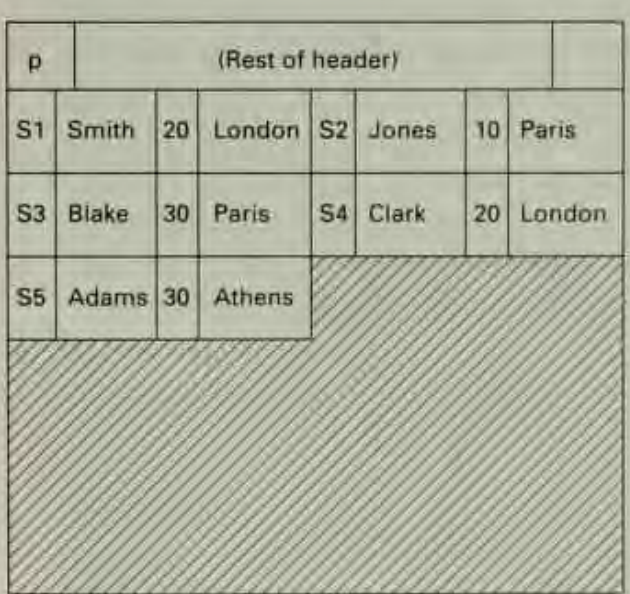

FIG. A.6 Layout of page *p* after initial loading of the five supplier records for S1–S5

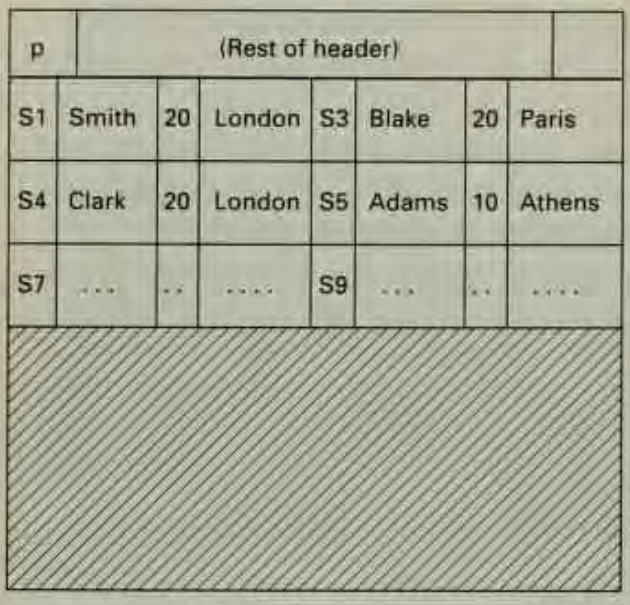

FIG. A.7 Layout of page *p* after inserting supplier S9, deleting supplier S2, and inserting supplier S7

record *r* consists of two parts—the page number of the page *p* containing *r*, and a byte offset from the foot of *p* identifying a slot that contains, in turn, the byte offset of *r* from the top of *p*. This scheme represents a good compromise between the speed of direct addressing and the flexibility of indirect addressing: Records can be shifted up and down within their containing page, as illustrated in Figs. A.7 and A.8, without having to change RIDs (only the local offsets at the foot of the page have to change); yet access to a given record given its RID is fast, involving only a single page access. (It is desir-

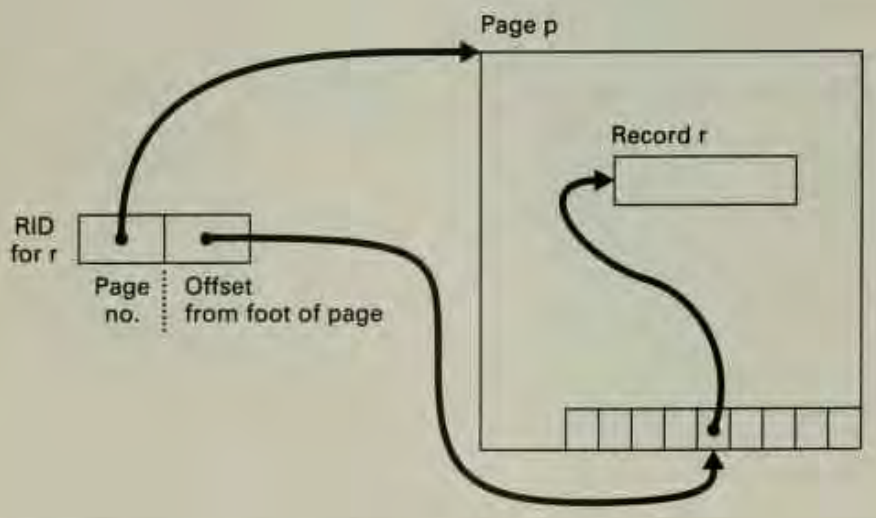

FIG. A.8 Implementation of stored record IDs (RIDs)

able that RIDs not change, because they are used elsewhere in the database as pointers to the records in question—for example, in indexes. If the RID of some record did in fact change, then all such pointer references elsewhere would have to be changed too.)

Note: Access to a specific stored record under the foregoing scheme might in rare cases involve two page accesses (but never more than two). Two accesses will be required if a varying length record is updated in such a way that it is now longer than it was before, and there is not enough free space on the page to accommodate the increase. In such a situation, the updated record will be placed on another page (an **overflow** page), and the original record will then be replaced by a pointer (another RID) to the new location. If the same thing happens again, so that the updated record has to be moved to still a third page, then the pointer in the original page will be changed to point to this newest location.

We are now almost ready to move on to our discussion of storage structures. From this point forward, we will assume for the most part (just as the DBMS normally assumes) that a given stored file is simply a collection of stored records, each uniquely identified by a record ID that never changes so long as that record remains in existence. A few final points to conclude this section:

- Note that one consequence of the preceding discussion is that, for any given stored file, it is *always* possible to access all of the stored records in that stored file sequentially—where by "sequentially" we mean "stored record sequence within page sequence within page set" (typically, ascending RID sequence). This sequence is often referred to loosely as **physical** sequence, though it should be clear that it does not necessarily correspond to any obvious physical sequence on the disk. For convenience, however, we will adopt the same term.
- Notice that access to a stored file in physical sequence is possible even if several files share the same page set (i.e., if stored files are interleaved). Records that do not belong to the stored file in question can simply be skipped over during the sequential scan (see the note below regarding record prefixes).
- It should be stressed that physical sequence is often at least adequate as an access path to a given stored file. Sometimes it might even be optimal (especially if the stored file is small, say not more than ten or so pages). However, it is frequently the case that something better is needed. And, as indicated in Section A.1, an enormous variety of techniques exist for achieving such a "something better."
- For the remainder of this appendix, we will usually assume for simplicity that the (unique) "physical" sequence for any given stored file is *primary key sequence* (as defined in Section A.3), barring any explicit statement to the contrary. Please note, however, that this assumption is made purely for the purpose of simplifying subsequent discussion; we recognize that there might be good reasons in practice for physically sequencing a given stored file in some other manner, e.g., by the value(s) of some other field(s), or simply by time of arrival (**chronological** sequence).
- For various reasons, a stored record will probably contain certain control information in addition to its user data fields. That information is typically collected to-

gether at the front of the record in the form of a **record prefix**. Examples of the kind of information found in such prefixes are the ID of the containing stored file (necessary if one page can contain records from several stored files), the record length (necessary for varying length records), a delete flag (necessary if records are not physically deleted at the time of a logical delete operation), pointers (necessary if records are chained together in any way), and so on. But of course all such control information will normally be concealed from the user (i.e., end user or application programmer).

- Finally, note that the user data fields in a given stored record will be of interest to the DBMS *but not to the file manager* (and not to the disk manager). The DBMS needs to be aware of those fields because it will use them as the basis for building indexes and the like. The file manager, however, has no need to be aware of them at all. Thus, another distinction between the DBMS and the file manager is that a given stored record has a known internal structure to the DBMS but is basically just a byte string to the file manager.

In the remainder of this appendix we describe some of the more important techniques for achieving the "something better" referred to above (i.e., an access path that is better than physical sequence). The techniques are discussed under the general headings of *indexing*, *hashing*, *pointer chains*, and *compression techniques*. One final general remark: The various techniques should not be seen as mutually exclusive. For example, it is perfectly feasible to have a stored file with (say) both hashed and indexed access to that file based on the same stored field, or with hashed access based on one field and pointer chain access based on another.

A.4 Indexing

Consider the suppliers table once again. Suppose the query "Find all suppliers in city *C*" (where *C* is a parameter) is an important one—i.e., one that is frequently executed and is therefore required to perform well. Given such a requirement, the DBA might choose the stored representation shown in Fig. A.9. In that representation, there are two stored files, a supplier file and a city file (probably in different page sets); the city file, which we assume to be stored in city sequence (because CITY is the primary key), includes pointers (RIDs) into the supplier file. To find all suppliers in London (say), the DBMS now has two possible strategies:

1. Search the entire supplier file, looking for all records with city value equal to London
2. Search the city file for the London entries, and for each such entry follow the pointer to the corresponding record in the supplier file

If the ratio of London suppliers to others is small, the second of these strategies is likely to be more efficient than the first, because (a) the DBMS is aware of the physical sequencing of the city file (it can stop its search of that file as soon as it finds a city that

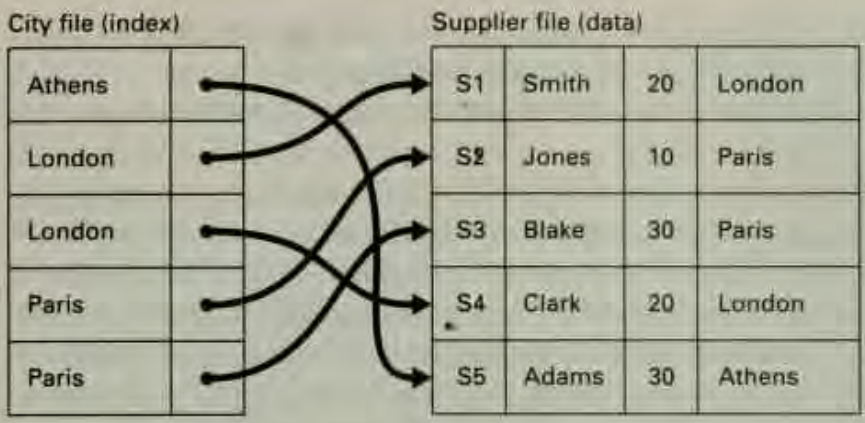

FIG. A.9 Indexing the supplier file on CITY

comes later than London in alphabetic order), and (b) even if it did have to search the entire city file, that search would still probably require fewer I/O's overall, because the city file is physically smaller than the supplier file (because the records are smaller).

In this example, the city file is said to be an **index** ("the CITY index") to the supplier file; equivalently, the supplier file is said to be *indexed by* the city file. An index is thus a special kind of stored file. To be specific, it is a file in which each entry (i.e., each record) consists of precisely two values, a data value and a pointer (RID); the data value is a value for some field of the indexed file, and the pointer identifies a record of that file that has that value for that field. The relevant field of the indexed file is called the *indexed field*, or sometimes the *index key* (we will not use this latter term, however).

Note: Indexes are so called by analogy with conventional book indexes, which also consist of entries containing "pointers" (page numbers) to facilitate the retrieval of information from an "indexed file" (i.e., the body of the book). Note, however, that unlike our CITY index, book indexes are *hierarchically compressed*—i.e., entries typically contain several page numbers, not just one. See Section A.7.

More terminology: An index on a primary key—e.g., an index on field S# of the supplier file—is sometimes called a *primary* index. An index on any other field—e.g., the CITY index in the example—is sometimes called a *secondary* index. Also, an index on a primary key, or more generally on any candidate key, is sometimes called a *unique* index.

How Indexes Are Used

The fundamental advantage of an index is that it speeds up retrieval. But there is a disadvantage too: It slows down updates. For instance, every time a new stored record is added to the indexed file, a new entry will also have to be added to the index. As a more specific example, consider what the DBMS must do to the CITY index of Fig. A.9 if supplier S2 moves from Paris to London. In general, therefore, the question that must be answered when some field is being considered as a candidate for indexing is:

Which is more important, efficient retrieval based on values of the field in question, or the update overhead involved in providing that efficient retrieval?

For the remainder of this section we concentrate on retrieval operations specifically.

Indexes can be used in essentially two different ways. First, they can be used for **sequential** access to the indexed file—where *sequential* means "in the sequence defined by values of the indexed field." For instance, the CITY index in the example above will allow records in the supplier file to be accessed in city sequence. Second, indexes can also be used for **direct** access to individual records in the indexed file on the basis of a given value for the indexed field. The query "Find suppliers in London" discussed at the start of the section illustrates this second case.

In fact, the two basic ideas just outlined can each be generalized slightly:

- *Sequential access:* The index can also help with *range* queries—for instance, "Find suppliers whose city is in some specified alphabetic range" (e.g., begins with a letter in the range L–R). Two important special cases are (a) "Find all suppliers whose city alphabetically precedes (or follows) some specified value," and (b) "Find all suppliers whose city is alphabetically first (or last)."
- *Direct access:* The index can also help with *list* queries—for instance, "Find suppliers whose city is in some specified list" (e.g., London, Paris, and New York).

In addition, there are certain queries—e.g., *existence tests*—that can be answered from the index alone, without any access to the indexed file at all. For example, consider the query "Are there any suppliers in Athens?" The response to this query is clearly "Yes" if and only if an entry for Athens exists in the CITY index.

A given stored file can have any number of indexes. For example, the supplier stored file might have both a CITY index and a STATUS index (see Fig. A.10). Those indexes could then be used to provide efficient access to supplier records on the basis of given values for either *or both* of CITY and STATUS. As an illustration of the "both" case, consider the query "Find suppliers in Paris with status 30." The CITY

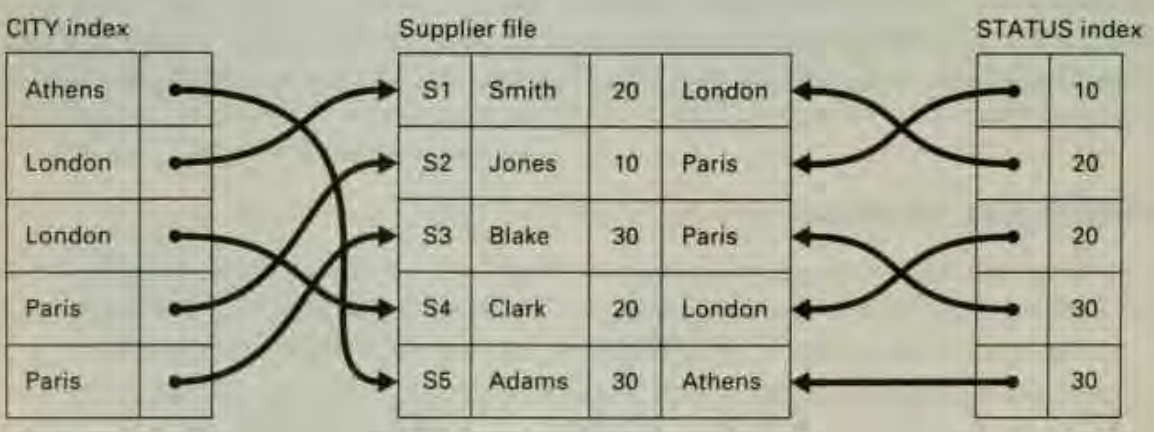

FIG. A.10 Indexing the supplier file on both CITY and STATUS

index gives the RIDs—*r2* and *r3*, say—for the suppliers in Paris; likewise, the STATUS index gives the RIDs—*r3* and *r5*, say—for suppliers with status 30. From these two sets of RIDs it is clear that the only supplier satisfying the original query is the supplier with RID equal to *r3* (namely, supplier S3). Only then does the DBMS have to access the supplier file itself, in order to retrieve the desired record.

More terminology: Indexes are sometimes referred to as **inverted lists,** for the following reason. First, a "normal" file—the supplier file of Figs. A.10 and A.11 can be taken as a typical "normal file"—lists, for each record, the values of the fields in that record. By contrast, an index lists, for each value of the indexed field, the records that contain that value. (The inverted-list database systems mentioned briefly at the end of Chapter 1 draw their name from this terminology.) And one more term: A file with an index on every field is sometimes said to be *fully inverted.*

Indexing on Field Combinations

It is also possible to construct an index on the basis of values of two or more fields in combination. For example, Fig. A.11 shows an index to the supplier file on the combination of fields CITY and STATUS, in that order. With such an index, the DBMS could respond to the query discussed above—"Find suppliers in Paris with status 30"—in a single scan *of a single index.* If the combined index were replaced by two separate indexes, then (as described earlier) that query would involve two separate index scans. Furthermore, it might be difficult in that case to decide which of those two scans should be done first; since the two possible sequences might have very different performance characteristics, the choice could be significant.

Note that the combined CITY/STATUS index can also serve as an index on the CITY field alone, since all the entries for a given city are at least still consecutive within the combined index. (Another, separate index will have to be provided if indexing on STATUS is also required, however.) In general, an index on the combination of fields

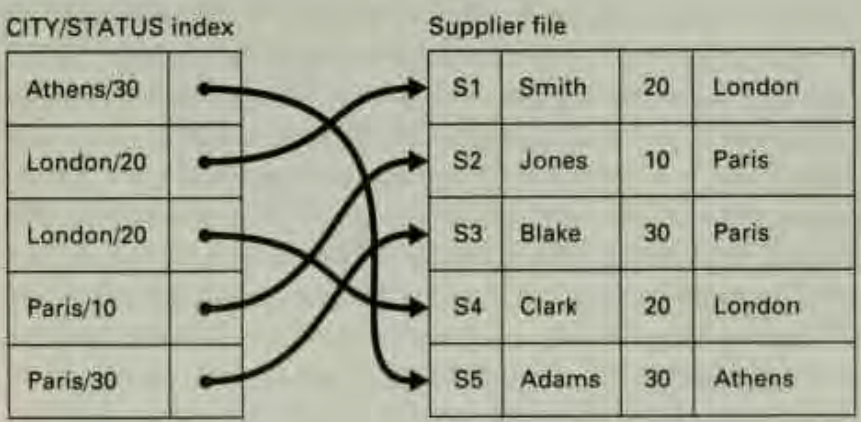

FIG. A.11 Indexing the supplier file on the combination of CITY and STATUS

F1, F2, F3, . . . Fn (in that order) will also serve as an index on *F1* alone, as an index on the combination *F1F2* (or *F2F1*), as an index on the combination *F1F2F3* (in any order), and so on. Thus the total number of indexes required to provide complete indexing in this way is not as large as might appear at first glance (see Exercise A.9 at the end of this appendix).

Dense *vs.* Nondense Indexing

As stated several times already, the fundamental purpose of an index is to speed up retrieval—more specifically, to reduce the number of disk I/O's needed to retrieve some given stored record. Basically, this purpose is achieved by means of *pointers;* and up to this point we have assumed that all such pointers are *record* pointers (i.e., RIDs). In fact, however, it would be sufficient for the stated purpose if those pointers were simply *page* pointers (i.e., page numbers). It is true that to find the desired record within a given page, the system would then have to do some additional work to search through the page in main memory, but the number of I/O's would remain unchanged. *Note:* As a matter of fact, the book index analogy mentioned earlier provides an example of an index in which the pointers are page pointers rather than "record" pointers.

We can take this idea further. Recall that any given stored file has a single "physical" sequence, represented by the combination of (a) the sequence of stored records within each page and (b) the sequence of pages within the containing page set. Suppose the supplier file is stored such that its physical sequence corresponds to the logical sequence as defined by the values of some field, say the supplier number field; in other words, the supplier file is *clustered* on that field (see the discussion of intra-file clustering at the end of Section A.2). Suppose also that an index is required on that field. Then there is no need for that index to include an entry for every stored record in the indexed file (i.e., the supplier file, in the example); all that is needed is an entry for each *page,* giving the highest supplier number on the page and the corresponding page number. See Fig. A.12 (where we assume for simplicity that a given page can hold a maximum of two supplier records).

As an example, consider what is involved in retrieving supplier S3 using this index. First the system must scan the index, looking for the first entry with supplier number greater than or equal to S3. It finds the index entry for supplier S4, which points to page *p* (say). It then retrieves page *p* and scans it in main memory, looking for the required stored record (which in this example, of course, will be found very quickly).

An index such as that of Fig. A.12 is said to be **nondense** (or sometimes **sparse**), because it does not contain an entry for every stored record in the indexed file. (By contrast, all indexes discussed prior to this point have been **dense**.) One advantage of a nondense index is that it will occupy less storage than a corresponding dense index, for the obvious reason that it contains fewer entries. As a result, it will probably be quicker to scan also. A disadvantage is that it might no longer be possible to perform existence tests on the basis of the index alone (see the brief note on this topic in the subsection "How indexes are used" earlier in this section).

Note that in general a given stored file can have at most one nondense index,

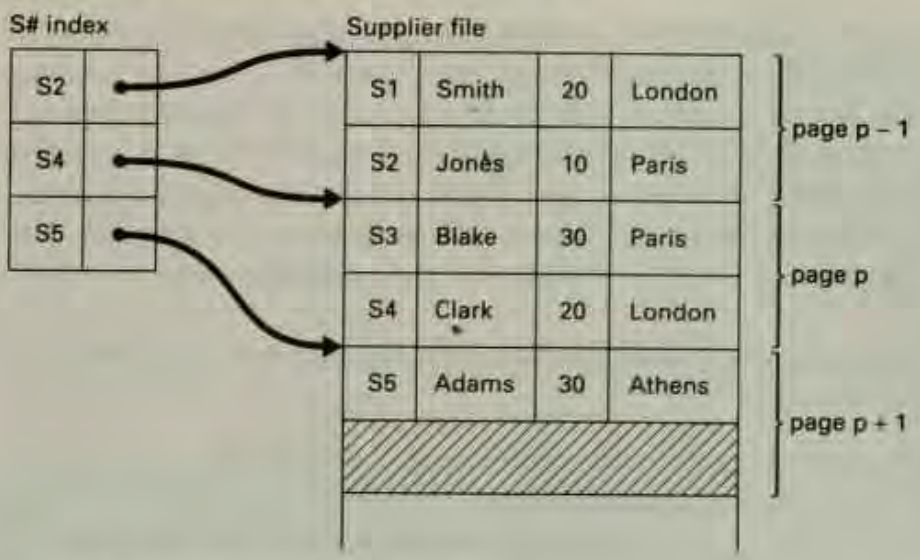

FIG. A.12 Example of a nondense index

because such an index relies on the (unique) physical sequence of the file in question. All other indexes must necessarily be dense.

B-trees

A particularly common and important kind of index is the **B-tree**. Although it is true that there is no single storage structure that is optimal for all applications, there is little doubt that if a single structure must be chosen, then B-trees of one kind or another are probably the one to choose. B-trees do generally seem to be the best all-around performer. For this reason, in fact, most relational systems support B-trees as their principal form of storage structure, and several support no other.

Before we can explain what a B-tree is, we must first introduce one more preliminary notion, namely the notion of a **multi-level** or **tree-structured** index.

The reason for providing an index in the first place is to remove the need for physical sequential scanning of the indexed file. However, physical sequential scanning is still needed in the *index*. If the indexed file is very large, then the index can itself get to be quite sizable, and sequentially scanning the index can itself get to be quite time-consuming. The solution to this problem is the same as before: We treat the index simply as a regular stored file, and build an index to it (an index to the index). This idea can be carried to as many levels as desired (three are common in practice; a file would have to be very large indeed to require more than three levels of indexing). Each level of the index acts as a nondense index to the level below (it *must* be nondense, of course, for otherwise nothing would be achieved—level n would contain the same number of entries as level $(n+1)$, and so would take just as long to scan).

Now we can discuss B-trees. A B-tree is a particular type of tree-structured index. B-trees as such were first described in a paper by Bayer and McCreight in 1972 [A.16]. Since that time, numerous variations on the basic idea have been proposed, by Bayer

himself and by many other investigators; as already suggested, B-trees of one kind or another are now probably the commonest storage structure of all in modern database systems. Here we describe the variation given by Knuth [A.1]. (We remark in passing that the index structure of IBM's "Virtual Storage Access Method" VSAM [A.18] is very similar to Knuth's structure; however, the VSAM version was invented independently and includes additional features of its own, such as the use of compression techniques. In fact, a precursor of the VSAM structure was described as early as 1969 [A.19].)

In Knuth's variation, the index consists of two parts, the *sequence set* and the *index set* (to use VSAM terminology).

- The **sequence set** consists of a single-level index to the actual data; that index is normally dense, but could be nondense if the indexed file were clustered on the indexed field. The entries in the index are (of course) grouped into pages, and the pages are (of course) chained together, such that the logical ordering represented by the index is obtained by taking the entries in physical order in the first page on the chain, followed by the entries in physical order in the second page on the chain, and so on. Thus the sequence set provides fast *sequential* access to the indexed data.
- The **index set,** in turn, provides fast *direct* access to the sequence set (and hence to the data too). The index set is actually a tree-structured index to the sequence set; in fact, it is the index set that is the real B-tree, strictly speaking. The combination of index set and sequence set is sometimes called a "B-plus"-tree (B^+-tree). The top level of the index set consists of a single node (i.e., a single page, but of course containing many index entries, like all the other nodes). That top node is called the **root**.

A simple example is shown in Fig. A.13, which we explain as follows. First, the values 6, 8, 12, . . . 97, 99 are values of the indexed field, *F* say. Consider the top node, which consists of two *F* values (50 and 82) and three pointers (actually page numbers). Data records with *F* less than or equal to 50 can be found (eventually) by following the left pointer from this node; similarly, records with *F* greater than 50 and less than or equal to 82 can be found by following the middle pointer; and records with *F* greater than 82 can be found by following the right pointer. The other nodes of the index set are interpreted analogously; note that (for example) following the right pointer from the first node at the second level takes us to all records with *F* greater than 32 *and also less than or equal to 50* (by virtue of the fact that we have already followed the left pointer from the next higher node).

The B-tree (i.e., index set) of Fig. A.13 is somewhat unrealistic, however, for the following two reasons:

- First, the nodes of a B-tree do not normally all contain the same number of data values;
- Second, they normally do contain a certain amount of free space.

In general, a "B-tree of order *n*" has at least *n* but not more than $2n$ data values at any given node (and if it has *k* data values, then it also has $k+1$ pointers). No data value

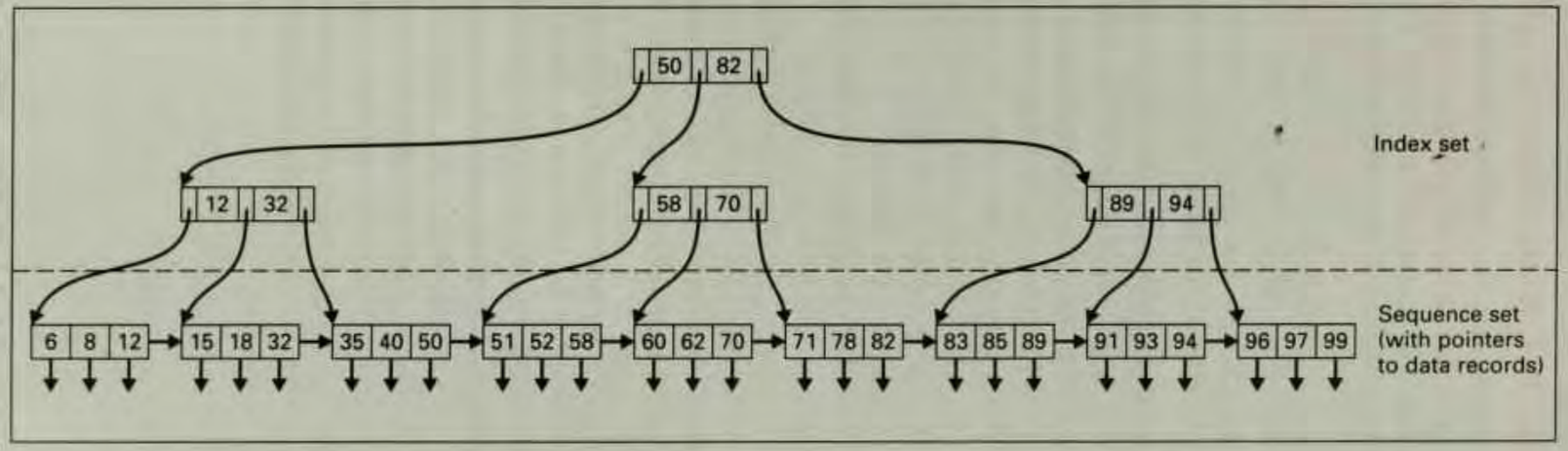

FIG. A.13 Part of a simple B-tree (Knuth's variation)

appears in the tree more than once. We give the algorithm for searching for a particular value *V* in the structure of Fig. A.13; the algorithm for the general B-tree of order *n* is a simple generalization.

```
set N to the root node ;
do until N is a sequence-set node ;
   let X, Y (X < Y) be the data values in node N ;
   if V ≤ X      then set N to the left   lower node of N ;
   if X < V ≤ Y then set N to the middle lower node of N ;
   if V > Y      then set N to the right  lower node of N ;
end ;
if V occurs in node N then exit /* found */ ;
if V does not occur in node N then exit /* not found */ ;
```

A problem with tree structures in general is that insertions and deletions can cause the tree to become *unbalanced*. A tree is unbalanced if the leaf nodes are not all at the same level—i.e., if different leaf nodes are at different distances from the root node. Since searching the tree involves a disk access for every node visited, search times can become very unpredictable in an unbalanced tree. *Note:* In practice, the top level of the index—possibly portions of other levels too—will typically be kept in main memory most of the time, which will have the effect of reducing the average number of disk accesses. The overall point remains valid, however.

The notable advantage of B-trees, by contrast, is that the B-tree insertion/deletion algorithm guarantees that the tree will always be balanced. (The "B" in "B-tree" is sometimes said to stand for "balanced" for this reason.) We briefly consider insertion of a new value, *V* say, into a B-tree of order *n*. The algorithm as described caters for the index set only, since (as explained earlier) it is the index set that is the B-tree proper; a trivial extension is needed to deal with the sequence set also.

- First, the search algorithm is executed to locate, not the sequence set node, but that node (*N* say) at the lowest level of the index set in which *V* logically belongs. If *N* contains free space, *V* is inserted into *N* and the process terminates.
- Otherwise, node *N* (which must therefore contain $2n$ values) is *split* into two nodes *N1* and *N2*. Let *S* be the original $2n$ values plus the new value *V*, in their logical sequence. The lowest *n* values in *S* are placed in the left node *N1*, the highest *n* values in *S* are placed in the right node *N2*, and the middle value, *W* say, is promoted to the parent node of *N*, *P* say, to serve as a separator value for nodes *N1* and *N2*. Future searches for a value *U*, on reaching node *P*, will be directed to node *N1* if $U \leq W$ and to node *N2* if $U > W$.
- An attempt is now made to insert *W* into *P*, and the process is repeated.

In the worst case, splitting will occur all the way to the top of the tree; a new root node (parent to the old root, which will now have been split into two) will be created, and the tree will increase in height by one level (but even so will still remain balanced).

The deletion algorithm is of course essentially the inverse of the insertion algorithm just described. Changing a value is handled by deleting the old value and inserting the new one.

A.5 Hashing

Hashing (also called **hash-addressing,** and sometimes—confusingly—**hash indexing**) is a technique for providing fast *direct* access to a specific stored record on the basis of a given value for some field. The field in question is usually but not necessarily the primary key. In outline, the technique works as follows.

- Each stored record is placed in the database at a location whose address (RID, or perhaps just page number) is computed as some function (the **hash function**) of some field of that record (the *hash field,* or sometimes *hash key;* we will not use this latter term, however). The computed address is called the **hash address**.
- To store the record initially, the DBMS computes the hash address for the new record and instructs the file manager to place the record at that position.
- To retrieve the record subsequently given the hash field value, the DBMS performs the same computation as before and instructs the file manager to fetch the record at the computed position.

As a simple illustration, suppose (a) that supplier number values are S100, S200, S300, S400, S500 (instead of S1, S2, S3, S4, S5), and (b) that each stored supplier record requires an entire page to itself, and consider the following hash function:

```
hash address (i.e., page number) =
   remainder after dividing numeric part of S# value by 13
```

This is a trivial example of a very common class of hash function called **division/remainder**. (For reasons that are beyond the scope of this appendix, the divisor in a division/remainder hash is usually chosen to be prime, as in our example.) The page numbers for the five suppliers are then 9, 5, 1, 10, 6, respectively, giving us the representation shown in Fig. A.14.

It should be clear from the foregoing description that hashing differs from indexing inasmuch as, while a given stored file can have any number of indexes, it can have *at most one* hash structure. To state this differently: A file can have any number of indexed fields, but only one hash field. (These remarks assume the hash is *direct*. By contrast, a file can have any number of *indirect* hashes. See reference [A.24].)

In addition to showing how hashing works, the example also shows why the hash function is necessary. It would theoretically be possible to use an "identity" hash function—e.g., to take the primary key value directly as the hash address (assuming, of course, that the primary key is numeric). Such a technique would generally be inadequate in practice, however, because the range of possible primary key values will usually be much wider than the range of available addresses. For instance, suppose that supplier numbers are in fact in the range S000–S999, as in the example above. Then there would be 1000 possible distinct supplier numbers, whereas there might in fact be only ten or so actual suppliers. Thus, in order to avoid a considerable waste of storage space, we would ideally like to find a hash function that would reduce any value in the range 000–999 to one in the range 0–9 (say). To allow a little room for future growth,

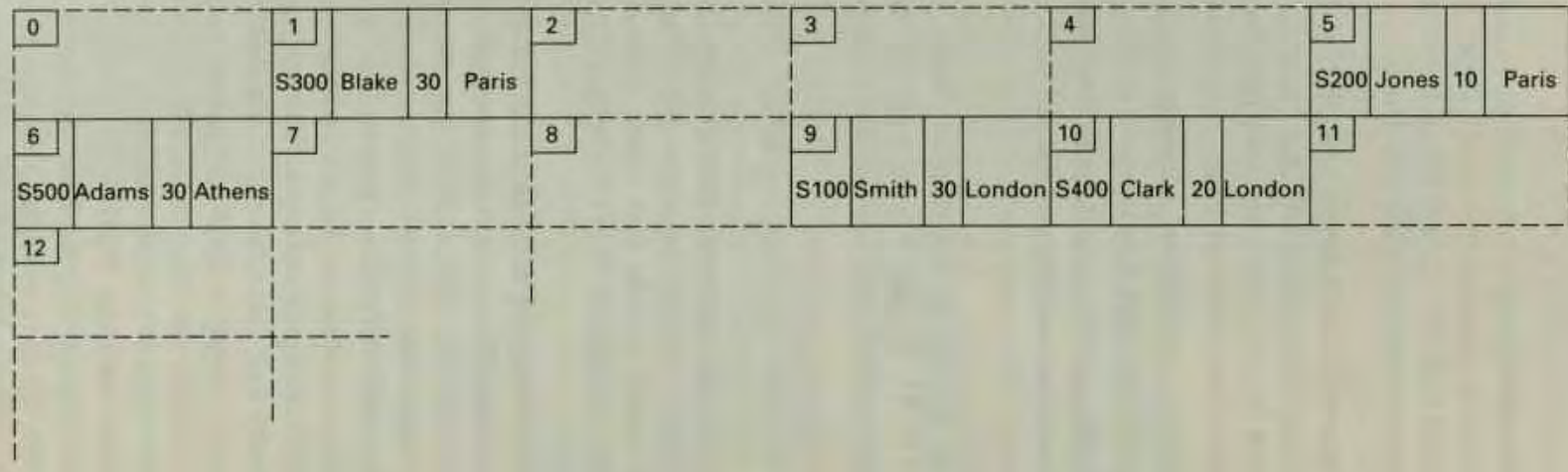

FIG. A.14 Example of a hashed structure

it is usual to extend the target range by 20 percent or so; that was why we chose a function that generated values in the range 0–12 rather than 0–9 in our example above.

The example also illustrates one of the disadvantages of hashing: The "physical sequence" of records within the stored file will almost certainly not be the primary key sequence, nor indeed any other sequence that has any sensible logical interpretation. (In addition, there may be gaps of arbitrary size between consecutive records.) In fact, the physical sequence of a stored file with a hashed structure is usually—not invariably—considered to represent no particular logical sequence.

(Of course, it is always possible to *impose* any desired logical sequence on a hashed file by means of an index; indeed, it is possible to impose several such sequences, by means of several indexes, one for each sequence. See also references [A.35] and [A.37], which discuss the possibility of hashing schemes that do preserve logical sequence in the file as stored.)

Another disadvantage of hashing in general is that there is always the possibility of **collisions**—that is, two or more distinct records ("synonyms") that hash to the same address. For example, suppose the supplier file (with suppliers S100, S200, etc.) also includes a supplier with supplier number S1400. Given the "divide by 13" hash function discussed above, that supplier would collide (at hash address 9) with supplier S100. The hash function as it stands is thus clearly inadequate—it needs to be extended somehow to deal with the collision problem.

In terms of our original example, one possible extension is to treat the remainder after division by 13, not as the hash address *per se*, but rather as the start point for a sequential scan. Thus, to insert supplier S1400 (assuming that suppliers S100–S500 already exist), we go to page 9 and search forward from that position for the first free page. The new supplier will be stored on page 11. To retrieve that supplier subsequently, we go through a similar procedure. This **linear search** method might well be adequate if (as is likely in practice) several records are stored on each page. Suppose each page can hold n stored records. Then the first n collisions at some hash address p will all be stored on page p, and a linear search through those collisions will be totally contained within that page. However, the next—i.e., $(n+1)$st—collision will of course have to be stored on some distinct **overflow** page, and another I/O will be needed.

Another approach to the collision problem, perhaps more frequently encountered in real systems, is to treat the result from the hash function, a say, as the storage address, not for a data record, but rather for an **anchor point**. The anchor point at storage address a is then taken as the head of a chain of pointers (a **collision chain**) linking together all records—or all pages of records—that collide at a. Within any given collision chain, the collisions will typically be kept in hash field sequence, to simplify subsequent searching.

Extendable Hashing

Yet another disadvantage of hashing as described above is that as the size of the hashed file increases, so the number of collisions also tends to increase, and hence the average access time increases correspondingly (because more and more time is spent searching

through sets of collisions). Eventually a point might be reached where it becomes desirable to reorganize the file—i.e., to unload the existing file and to reload it, using a new hashing function.

Extendable hashing [A.28] is a nice variation on the basic hashing idea that alleviates the foregoing problems. In fact, extendable hashing guarantees that the number of disk accesses needed to locate a specific record (i.e., the record having a specific primary key value) is never more than two, and will usually be only one, no matter what the file size might be. (It therefore also guarantees that file reorganization will never be required.) *Note:* Values of the hash field must be unique in the extendable hashing scheme, which of course they will be if that field is in fact the primary key as suggested at the start of this section.

In outline, the scheme works as follows.

1. Let the basic hash function be h, and let the primary key value of some specific record r be k. Hashing k—i.e., evaluating $h(k)$—yields a value s called the *pseudokey* of r. Pseudokeys are not interpreted directly as addresses but instead lead to storage locations in an indirect fashion as described below.

2. The stored file has a *directory* associated with it, also stored on the disk. The directory consists of a header, containing a value d called the *depth* of the directory, together with 2^d pointers. The pointers are pointers to data pages, which contain the actual stored records (many records per page). A directory of depth d can thus handle a maximum file size of 2^d distinct data pages.

3. If we consider the leading d bits of a pseudokey as an unsigned binary integer b, then the ith pointer in the directory ($1 \leq i \leq 2^d$) points to a page that contains all records for which b takes the value $i-1$. In other words, the first pointer points to the page containing all records for which b is all zeros, the second pointer points to the page for which b is 0 . . . 01, and so on. (These 2^d pointers are typically not all distinct; that is, there will typically be fewer than 2^d distinct data pages. See Fig. A.15.) Thus, to find the record having primary key value k, we hash k to find the pseudokey s and take the first d bits of that pseudokey; if those bits have the numeric value $i-1$, we go to the ith pointer in the directory (first disk access) and follow it to the page containing the required record (second disk access).

   Note: In practice the directory will usually be sufficiently small that it can be kept in main memory most of the time. Thus the "two" disk accesses will usually reduce to one in practice.

4. Each data page also has a header giving the *local depth* p of that page ($p \leq d$). Suppose, for example, that d is three, and that the first pointer in the directory (the 000 pointer) points to a page for which the local depth p is two. Local depth two here means that, not only does this page contain all records with pseudokeys starting 000, it contains *all* records with pseudokeys starting 00 (i.e., those starting 000 and also those starting 001). In other words, the 001 directory pointer also points to this page. Again, see Fig. A.15.

5. Continuing the example from paragraph 3 above, suppose now that the 000 data page is full and we wish to insert a new record having a pseudokey that starts 000

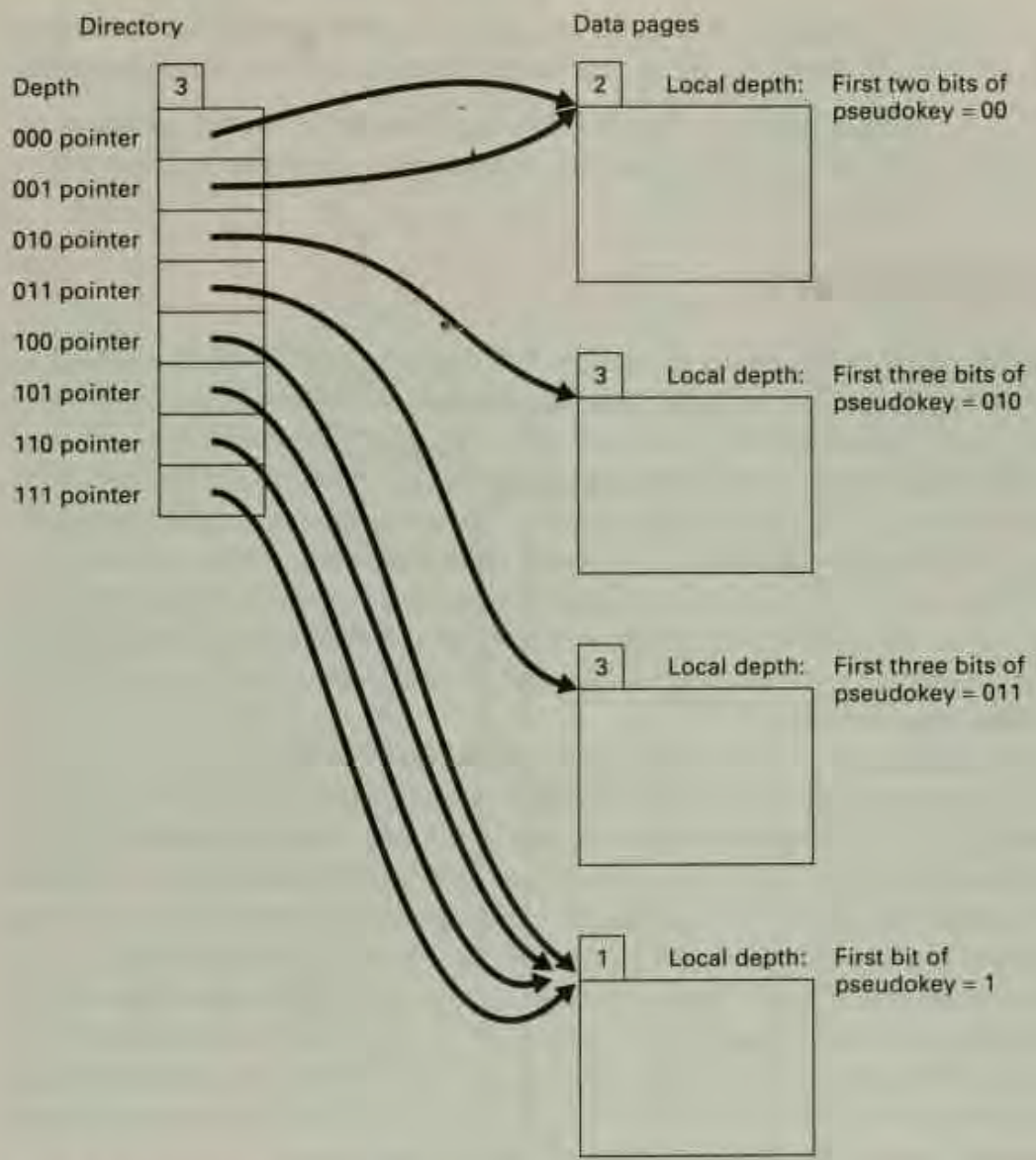

FIG. A.15 Example of extendable hashing

(or 001). At this point the page is split in two; that is, a new, empty page is acquired, and all 001 records are moved out of the old page and into the new one. The 001 pointer in the directory is changed to point to the new page (the 000 pointer still points to the old one). The local depth p for each of the two pages will now be three, not two.

6. Again continuing the example, suppose that the data page for 000 becomes full again and has to split again. The existing directory cannot handle such a split, because the local depth of the page to be split is already equal to the directory depth. Therefore we *double the directory;* that is, we increase d by one and replace each pointer by a pair of adjacent, identical pointers. The data page can now be split; 0000 records are left in the old page and 0001 records go in the new page; the first pointer in the directory is left unchanged (i.e., it still points to the old page), the

second pointer is changed to point to the new page. Note that doubling the directory is a fairly inexpensive operation, since it does not involve access to any of the data pages.

So much for our discussion of extendable hashing. Numerous further variations on the basic hashing idea have been devised; see, for example, references [A.29–A.36].

A.6 Pointer Chains

Suppose again, as at the beginning of Section A.4, that the query "Find all suppliers in city *C*" is an important one. Another stored representation that can handle that query reasonably well—possibly better than an index, though not necessarily so—uses *pointer chains*. Such a representation is illustrated in Fig. A.16. As can be seen, it involves two stored files, a supplier file and a city file, much as in the index representation of Fig. A.9 (this time both files are probably in the same page set, for reasons to be explained in Section A.7). In the pointer chain representation of Fig. A.16, however, the city file is not an index but what is sometimes referred to as a *parent* file. The supplier file is accordingly referred to as the *child* file, and the overall structure is an example of **parent/child organization**.

In the example, the parent/child structure is based on supplier city values. The parent (city) file contains one stored record for each distinct supplier city, giving the city value and acting as the head of a *chain* or *ring* of pointers linking together all child (supplier) records for suppliers in that city. Note that the city field has been removed from the supplier file; to find all suppliers in London (say), the DBMS can search the city file for the London entry and then follow the corresponding pointer chain.

The principal advantage of the parent/child structure is that the insert/delete algorithms are somewhat simpler, and might conceivably be more efficient, than the corresponding algorithms for an index; also, the structure will probably occupy less storage than the corresponding index structure, because each city value appears exactly once instead of many times. The principal disadvantages are as follows:

- For a given city, the only way to access the *n*th supplier is to follow the chain and access the 1st, 2nd, . . . $(n - 1)$st supplier too. Unless the supplier records are appropriately clustered, each access will involve a separate seek operation, and the time taken to access the *n*th supplier could be considerable.
- Although the structure might be suitable for the query "Find suppliers in a given city," it is of no help—in fact, it is a positive hindrance—for the converse query "Find the city for a given supplier" (where the given supplier is identified by a given supplier number). For this latter query, either a hash or an index on the supplier file is probably desirable; note that a parent/child structure based on supplier numbers would not make much sense (why not?). And even when the given supplier record has been located, it is still necessary to follow the chain to the parent record to discover the desired city (the need for this extra step is our justification for claiming that the parent/child structure is actually a hindrance for this class of query).

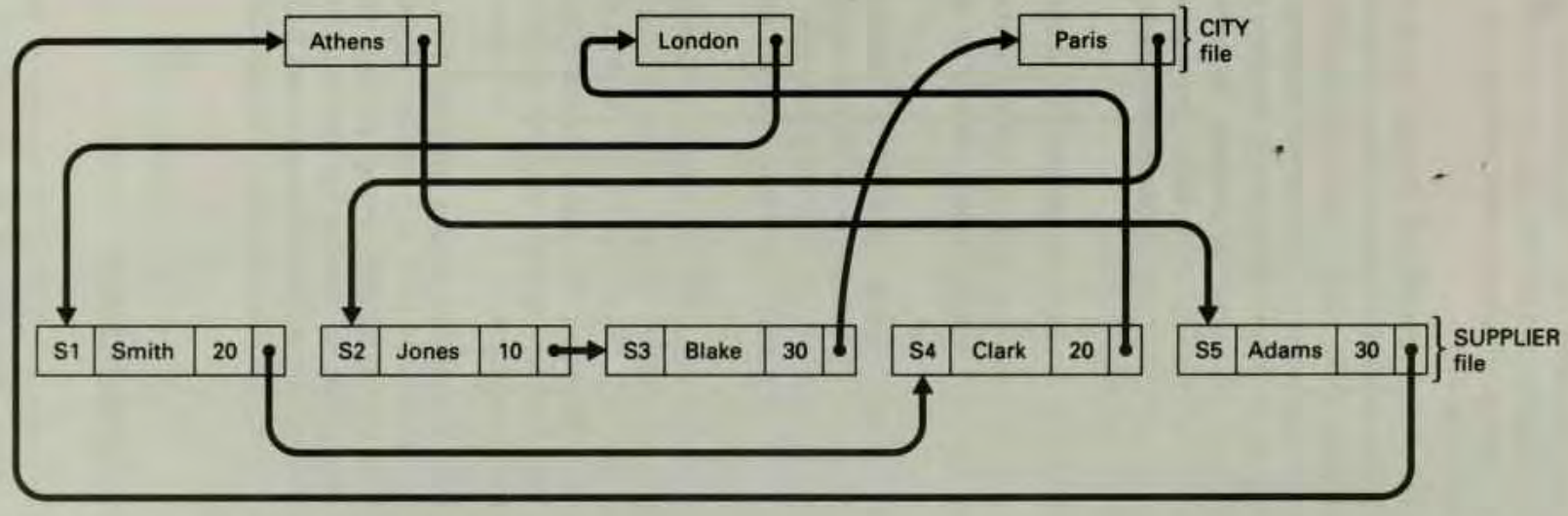

FIG. A.16 Example of a parent/child structure

Note, moreover, that the parent (city) file will probably require index or hash access too if it is of any significant size. Hence pointer chains alone are not really an adequate basis for a storage structure—other mechanisms such as indexes will almost certainly be needed as well.

- Because (a) the pointer chains actually run through the stored records (i.e., the record prefixes physically include the relevant pointers), and (b) values of the relevant field are factored out of the child records and placed in the parent records instead, it is a nontrivial task to create a parent/child structure over an existing set of records. In fact, such an operation will typically require a database reorganization, at least for the relevant portion of the database. By contrast, it is a comparatively straightforward matter to create a new index over an existing set of records. *Note:* Creating a new hash will also typically require a reorganization, incidentally, unless the hash is indirect [A.24].

Several variations are possible on the basic parent/child structure. For example:

- The pointers could be made two-way. One advantage of this variation is that it simplifies the process of pointer adjustment necessitated by the operation of deleting a child record.
- Another extension would be to include a pointer (a "parent pointer") from each child record direct to the corresponding parent; this extension would reduce the amount of chain-traversing involved in answering the query "Find the city for a given supplier" (note, however, that it does not eliminate the need for a hash or index to help with that query).
- Yet another variation would be *not* to remove the city field from the supplier file but to repeat the field in the supplier records (a simple form of controlled redundancy). Certain retrievals—e.g., "Find the city for supplier S4"—would then become more efficient. Note, however, that that increased efficiency has nothing to do with the pointer chain structure as such, also that a hash or index on supplier numbers will probably still be required.

Finally, of course, just as it is possible to have any number of indexes over a given stored file, so it is equally possible to have any number of pointer chains running through a given stored file. (It is also possible, though perhaps unusual, to have both.) Fig. A.17 shows a representation for the supplier file that involves two distinct pointer chains, and therefore two distinct parent/child structures, one with a city file as parent (as in Fig. A.16) and one with a status file as parent. The supplier file is the child file for both of these structures.

A.7 Compression Techniques

Compression techniques are ways of reducing the amount of storage required for a given collection of stored data. Quite frequently the result of such compression will be not only to save on storage space, but also (and probably more significantly) to save on

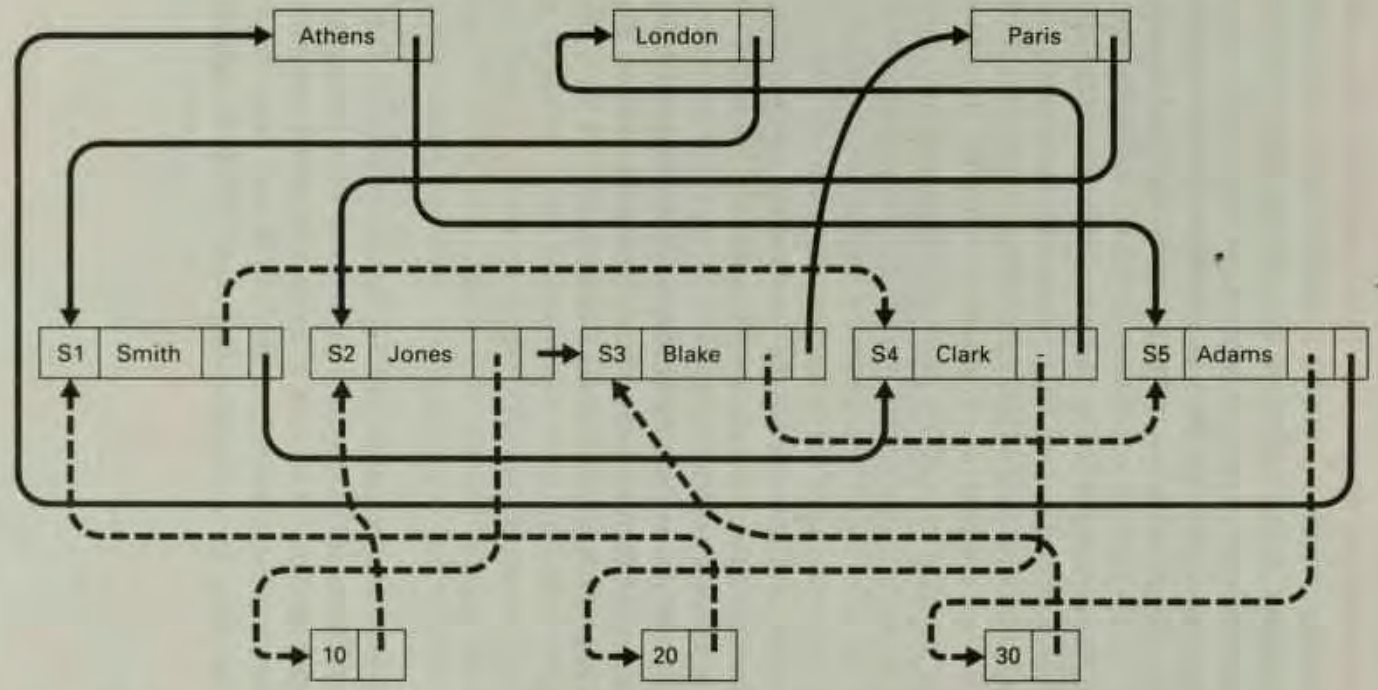

FIG. A.17 Example of a multiple parent/child organization

disk I/O; for if the data occupies less space, then fewer I/O operations will be needed to access it. On the other hand, extra processing will be needed to decompress the data after it has been retrieved. On balance, however, the I/O savings will probably outweigh the disadvantage of that additional processing.

Compression techniques are designed to exploit the fact that data values are almost never completely random but instead display a certain amount of predictability. As a trivial example, if a given person's name in a name-and-address file starts with the letter R, then it is extremely likely that the next person's name will start with the letter R also—assuming, of course, that the file is in alphabetical order by name.

A common compression technique is thus to replace each individual data value by some representation of the difference between it and the value that immediately precedes it—**differential compression**. Note, however, that such a technique requires that the data in question be accessed sequentially, because to decompress any given stored value requires knowledge of the immediately preceding stored value. Differential compression thus has its main applicability in situations when the data must be accessed sequentially anyway, as in the case of (for example) the entries in a single-level index. Note moreover that, in the case of an index specifically, the pointers can be compressed as well as the data—for if the logical data ordering imposed by the index is the same as, or close to, the physical ordering of the underlying file, then successive pointer values in the index will be quite similar to one another, and pointer compression is likely to be beneficial. In fact, indexes almost always stand to gain from the use of compression, at least for the data if not for the pointers.

To illustrate differential compression, we depart for a moment from suppliers-and-parts and consider a page of entries from an "employee name" index. Suppose the first four entries on that page are for the following employees:

```
Roberton
Robertson
Robertstone
Robinson
```

Suppose also that employee names are 12 characters long, so that each of these names should be considered (in its uncompressed form) to be padded at the right with an appropriate number of blanks. One way to apply differential compression to this set of values is by replacing those characters at the front of each entry that are the same as those in the previous entry by a corresponding count: **front compression**. This approach yields:

```
0 - Roberton++++
6 - son+++
7 - tone+
3 - inson++++
```

(trailing blanks now shown explicitly as "+").

Another possible compression technique for this set of data is simply to eliminate all trailing blanks (again, replacing them by an appropriate count): an example of **rear compression**. Further rear compression can be achieved by dropping all characters to

the right of the one required to distinguish the entry in question from its two immediate neighbors, as follows:

```
0 - 7 - Roberto
6 - 2 - so
7 - 1 - t
3 - 1 - i
```

The first of the two counts in each entry here is as in the previous example, the second is a count of the number of characters recorded. We have assumed that the next entry does not have "Robi" as its first four characters when decompressed. Note, however, that we have actually lost some information from this index. That is, when decompressed, it looks like this:

```
Roberto?????
Robertso????
Robertst????
Robi????????
```

(where "?" represents an unknown character). Such a loss of information is obviously permissible only if the data is recorded in full *somewhere*—in the example, in the underlying employee file.

Hierarchic Compression

Suppose a given stored file is physically sequenced (i.e., clustered) by values of some stored field *F*, and suppose also that each distinct value of *F* occurs in several (consecutive) records of that file. For example, the supplier stored file might be clustered by values of the city field, in which case all London suppliers would be stored together, all Paris suppliers would be stored together, and so on. In such a situation, the set of all supplier records for a given city might profitably be compressed into a single **hierarchic** stored record, in which the city value in question appears exactly once, followed by supplier number, name, and status information for each supplier that happens to be located in that city. See Fig. A.18.

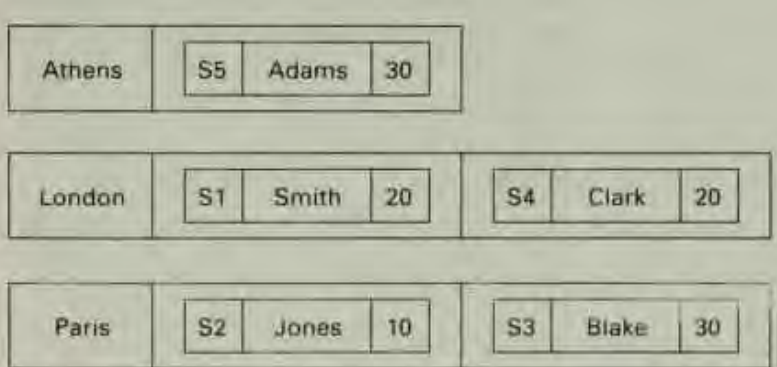

FIG. A.18 Example of hierarchic compression (intra-file)

The stored record *type* illustrated in Fig. A.18 consists of two parts: a fixed part, namely the city field, and a varying part, namely the set of supplier entries. (The latter part is varying in the sense that the number of entries it contains—i.e., the number of suppliers in the city in question—varies from one occurrence of the record to another.) As explained in Chapter 3, such a varying set of entries within a record is usually referred to as a **repeating group**. Thus we would say that the hierarchic record type of Fig. A.18 consists of a single city field and a repeating group of supplier information, and the repeating group in turn consists of a supplier number field, a supplier name field, and a supplier status field (one instance of the group for each supplier in the relevant city).

Hierarchic compression of the type just described is often particularly appropriate in an index, where it is commonly the case that several successive entries all have the same data value (but of course different pointer values).

It follows from the foregoing that hierarchic compression of the kind illustrated is feasible only if intra-file clustering is in effect. As the reader might already have realized, however, a similar kind of compression can be applied with *inter*-file clustering also. Suppose that suppliers and shipments are clustered as suggested at the end of Section A.2—that is, shipments for supplier S1 immediately follow the supplier record for S1, shipments for supplier S2 immediately follow the supplier record for S2, and so on. More specifically, suppose that supplier S1 and the shipments for supplier S1 are stored on page *p1*, supplier S2 and the shipments for supplier S2 are stored on page *p2*, etc. Then an inter-file compression technique can be applied as shown in Fig. A.19.

Note: Although we describe this example as "inter-file," it really amounts to combining the supplier and shipment files into a single file and then applying *intra*-file compression to that single file. Thus this case is not really different in kind from the case already illustrated in Fig. A.18.

We conclude this subsection by remarking that the pointer chain structure of Fig.

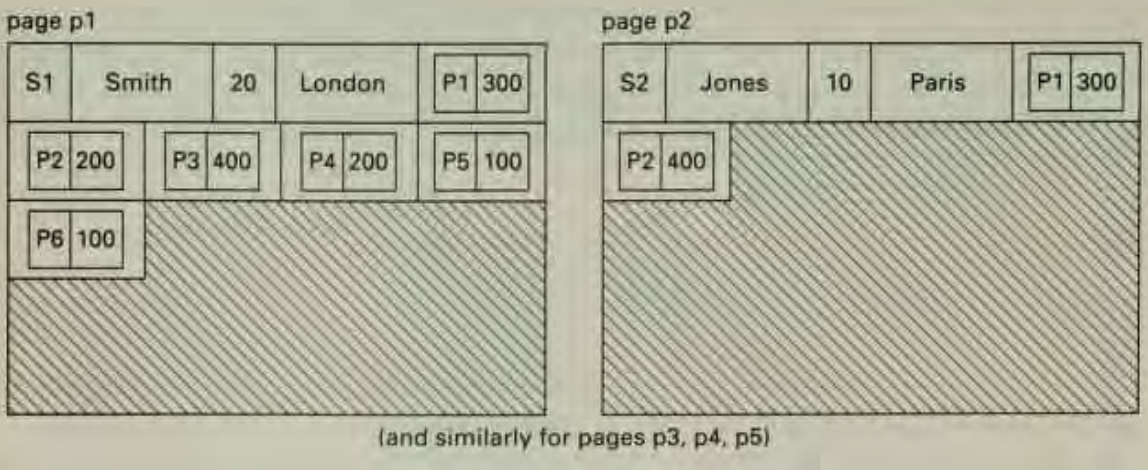

FIG. A.19 Example of hierarchic compression (inter-file)

A.16 can be regarded as a kind of inter-file compression that does not require a corresponding inter-file clustering (or, rather, the pointers provide the logical effect of such a clustering—so that compression is possible—but do not necessarily provide the corresponding physical performance advantage at the same time—so that compression, though possible, might not be a good idea).

Huffman Coding

"Huffman coding" [A.39] is a character encoding technique, little used in current systems, but one that can nevertheless result in significant data compression if different characters occur in the data with different frequencies (which is the normal situation, of course). The basic idea is as follows: Bit string encodings are assigned to represent characters in such a way that different characters are represented by bit strings of different lengths, and the most commonly occurring characters are represented by the shortest strings. Also, no character has an encoding (of *n* bits, say) such that those *n* bits are identical to the first *n* bits of some other character encoding.

As a simple example, suppose the data to be represented involves only the characters A, B, C, D, E, and suppose also that the relative frequency of occurrence of those five characters is as indicated in the following table:

Character	Frequency	Code
E	35%	1
A	30%	01
D	20%	001
C	10%	0001
B	5%	0000

Character E has the highest frequency and is therefore assigned the shortest code, a single bit, say a 1-bit. All other codes must then start with a 0-bit and must be at least two bits long (a lone 0-bit would not be valid, since it would be indistinguishable from the leading portion of other codes). Character A is assigned the next shortest code, say 01; all other codes must therefore begin 00. Similarly, characters D, C, and B are assigned codes 001, 0001, and 0000 respectively. *Exercise for the reader:* What English words do the following strings represent?

```
1. 00110001010011

2. 010001000110011
```

Given the encodings shown, the expected average length of a coded character, in bits, is

```
0.35 * 1 + 0.30 * 2 + 0.20 * 3 + 0.10 * 4 + 0.05 * 4 = 2.15  bits,
```

whereas if every character were assigned the same number of bits, as in a conventional character coding scheme, we would need three bits per character (to allow for the five possibilities).

A.8 Summary

In this appendix we have taken a lengthy—by no means exhaustive!—look at some of the most important storage structures used in current practice. We have also described in outline how the data access software typically functions, and have sketched the ways in which responsibility is divided up among the **DBMS**, the **file manager**, and the **disk manager**. Our purpose throughout has been to explain overall concepts, not to describe in fine detail how the various system components and storage structures actually work; indeed, we have tried hard not to get bogged down in too much detail, though of course a certain amount of detail is unavoidable.

By way of summary, here is a brief review some of the major topics we have touched on. First, we described **clustering**, the basic idea of which is that records that are used together should be stored physically close together. We also explained how stored records are identified internally by **record IDs** or **RIDs**. Then we considered some of the most important storage structures encountered in practice:

- **Indexes** (several variations thereof, including in particular **B-trees**) and their use for both **sequential** and **direct** access
- **Hashing** (including in particular **extendable** hashing) and its use for **direct** access
- **Pointer chains** (also known as **parent/child structures**) and numerous variations thereof

We also examined a variety of **compression** techniques.

We conclude by stressing the point that most users are (or should be) unconcerned with most of this material most of the time. The only "user" who needs to understand these ideas in detail is the DBA, who is responsible for the physical design of the database and for performance monitoring and tuning. For other users, such considerations should preferably all be under the covers, though it is probably true to say that those users will perform their job better if they have some idea of the way the system functions internally. For DBMS implementers, on the other hand, a working knowledge of this material (indeed, an understanding that goes much deeper than this introductory survey does) is clearly desirable, if not mandatory.

Exercises

Exercises A.1–A.8 might prove suitable as a basis for group discussion; they are intended to lead to a deeper understanding of various physical database design considerations. Exercises A.9 and A.10 have rather a mathematical flavor.

A.1 Investigate any database systems (the larger the better) that might be available to you. For each such system, identify the components that perform the functions ascribed in the body of this appendix to, respectively, the disk manager, the file manager, and the DBMS proper. What kind of disks or other media does the system support? What is the page size? What are the disk capacities, both theoretical (in bytes) and actual (in pages)? What are the data rates? The access times? How do those access times compare with the speed of main memory? Are there any limits on file size or database size? If so, what are they? Which of the storage structures de-

scribed in this appendix does the system support? Does it support any others? If so, what are they?

A.2 A company's personnel database is to contain information about the divisions, departments, and employees of that company. Each employee works in one department; each department is part of one division. Invent some sample data and sketch some possible corresponding storage structures. Where possible, state the relative advantages of each of those structures—i.e., consider how typical retrieval and update operations would be handled in each case. *Hint:* The constraints "each employee works in one department" and "each department is part of one division" are structurally similar to the constraint "each supplier is located in one city" (they are all examples of **many-to-one relationships**). A difference is that we would probably like to record more information in the database for departments and divisions than we did for cities.

A.3 Repeat Exercise A.2 for a database that is to contain information about customers and items. Each customer can order any number of items; each item can be ordered by any number of customers. *Hint:* There is a **many-to-many** relationship here between customers and items. One way to represent such a relationship is by means of a *double index*. A double index is an index that is used to index two data files simultaneously; a given entry in such an index corresponds to a pair of related data records, one from each of the two files, and contains two data values and two pointers. Can you think of any other ways of representing many-to-many(-to-many-...) relationships?

A.4 Repeat Exercise A.2 for a database that is to contain information about parts and components, where a component is itself a part and can have lower-level components. *Hint:* How does this problem differ from that of Exercise A.3?

A.5 A stored file of data records with no additional access structure (i.e., no index, no hash, etc.) is sometimes called a **heap**. New records are inserted into a heap wherever there happens to be room. For small files—certainly for any file not requiring more than (say) nine or ten pages of storage—a heap is probably the most efficient structure of all. Most files are bigger than that, however, and in practice all but the smallest files should have some additional access structure, say (at least) an index on the primary key. State the relative advantages and disadvantages of an indexed structure when compared with a heap structure.

A.6 We referred several times in the body of this appendix to physical clustering. For example, it might be advantageous to store the supplier records such that their physical sequence is the same as or close to their logical sequence as defined by values of the supplier number field (the *clustering field*). How can the DBMS provide such physical clustering?

A.7 In Section A.5 we suggested that one method of handling hash collisions would be to treat the output from the hash function as the start point for a sequential scan (the *linear search* technique). Can you see any difficulties with that scheme?

A.8 What are the relative advantages and disadvantages of the multiple parent/child organization? (See the end of Section A.6. It might help to review the advantages and disadvantages of the multiple index organization. What are the similarities? What are the differences?)

A.9 Let us define "complete indexing" to mean that an index exists for every distinct field combination in the indexed file. For example, complete indexing for a file with two fields A and B would require two indexes, one on the combination AB (in that order) and one on the combination BA (in that order). How many indexes are needed to provide complete indexing for a file defined on (a) 3 fields; (b) 4 fields; (c) N fields?

A.10 Consider a simplified B-tree (index set plus sequence set) in which the sequence set contains a pointer to each of N stored data records, and each level above the sequence set (i.e., each level of the index set) contains a pointer to every page in the level below. At the top (root) level, of course, there is a single page. Suppose also that each page of the index set contains n index entries. Derive expressions for the number of *levels* and the number of *pages* in the entire B-tree.

A.11 The first ten values of the indexed field in a particular indexed file are as follows:

```
Abrahams,GK
Ackermann,LZ
Ackroyd,S
Adams,T
Adams,TR
Adamson,CR
Allen,S
Ayres,ST
Bailey,TE
Baileyman,D
```

Each is padded with blanks at the right to a total length of 15 characters. Show the values actually recorded in the *index* if the front and rear compression techniques described in Section A.7 are applied. What is the percentage saving in space? Show the steps involved in retrieving (or attempting to retrieve) the stored records for "Ackroyd,S" and "Adams,V". Show also the steps involved in inserting a new stored record for "Allingham,M".

References and Bibliography

The following references are organized into groups, as follows. References [A.1–A.10] are textbooks that either are devoted entirely to the topic of this appendix or at least include a detailed treatment of it. References [A.11–A.15] describe some formal approaches to the subject. References [A.16–A.23] are concerned specifically with indexing, especially B-trees; references [A.24–A.38] represent a selection from the very extensive literature on hashing; references [A.39–A.40] discuss compression techniques; and, finally, references [A.41–A.59] address some miscellaneous storage structures and related issues (in particular, references [A.48–A.59] discuss certain new storage media and new kinds of applications, and new storage structures for those media and applications).

A.1 Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Reading, Mass.: Addison-Wesley (1973).

Volume III of Knuth's classic series of volumes contains a comprehensive analysis of search algorithms. For *database* searching, where the data resides in secondary storage, the most directly applicable sections are 6.2.4 (Multiway Trees), 6.4 (Hashing), and 6.5 (Retrieval on Secondary Keys).

A.2 James Martin. *Computer Data-Base Organization* (2nd edition). Englewood Cliffs, N.J.: Prentice-Hall (1977).

This book is divided into two major parts, "Logical Organization" and "Physical Organization." The latter part consists of an extensive description (well over 300 pages) of storage structures and corresponding access techniques.

A.3 Toby J. Teorey and James P. Fry. *Design of Database Structures*. Englewood Cliffs, N.J.: Prentice-Hall (1982).

(Same as reference [12.27].) A tutorial and handbook on database design, both physical and logical. Over 200 pages are devoted to physical design.

A.4 Gio Wiederhold. *Database Design* (2nd edition). New York, N.Y.: McGraw-Hill (1983).

This book (15 chapters) includes a good survey of secondary storage devices and their performance parameters (one chapter, nearly 50 pages), and an extensive analysis of secondary storage structures (four chapters, over 250 pages).

A.5 T. H. Merrett. *Relational Information Systems*. Reston, VA.: Reston Publishing Company, Inc. (1984).

Includes a lengthy introduction to, and analysis of, a variety of storage structures (about 100 pages), covering not only the structures described in the present appendix but also several others.

A.6 Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems: Volume I.* Rockville, MD.: Computer Science Press (1988).

Includes a treatment of storage structures that is rather more theoretical than that of the present appendix.

A.7 Henry F. Korth and Abraham Silberschatz. *Database System Concepts* (2nd edition). New York, N.Y.: McGraw-Hill (1991).

A.8 Peter D. Smith and G. Michael Barnes. *Files and Databases: An Introduction.* Reading, Mass.: Addison-Wesley (1987).

A.9 Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems* (2nd edition). Redwood City, Calif.: Benjamin/Cummings (1994).

References [A.7], [A.8], and [A.9] are all textbooks on database systems. Each includes material on storage structures that goes beyond the treatment in the present appendix in certain respects (extensively so, in the case of [A.8]).

A.10 Sakti P. Ghosh. *Data Base Organization for Data Management* (2nd edition). Orlando, Fla.: Academic Press (1986).

The primary emphasis of this book is on storage structures and associated access methods—of the book's ten chapters, at least six are devoted to these topics. The treatment is fairly abstract.

A.11 David K. Hsiao and Frank Harary. "A Formal System for Information Retrieval from Files." *CACM 13*, No. 2 (February 1970).

This paper represents what was probably the earliest attempt to unify the ideas of different storage structures—principally indexes and pointer chains—into a general model, thereby providing a basis for a formal theory of such structures. A generalized retrieval algorithm is presented for retrieving records from the general structure that satisfy an arbitrary Boolean combination of "*field* = *value*" conditions.

A.12 Dennis G. Severance. "Identifier Search Mechanisms: A Survey and Generalized Model." *ACM Comp. Surv. 6*, No. 3 (September 1974).

This paper falls into two parts. The first part provides a tutorial on certain storage structures—basically hashing and indexing. The second part has points in common with reference [A.11]; like that paper, it defines a unified structure, here called a *trie-tree* structure, that combines and generalizes ideas from the structures discussed in the first part. (The term *trie*, pronounced *try*, derives from a paper by Fredkin [A.42].) The resulting structure provides a general model that can represent a wide variety of different structures in terms of a small number of parameters; it can therefore be used (and has in fact been used) to help in choosing a particular structure during the process of physical database design.

A difference between this paper and reference [A.11] is that the trie-tree structure handles hashes but not pointer chains, whereas the proposal of reference [A.11] handles pointer chains but not hashes.

A.13 M. E. Senko, E. B. Altman, M. M. Astrahan, and P. L. Fehder. "Data Structures and Accessing in Data-Base Systems." *IBM Sys. J. 12*, No. 1 (1973).

This paper is in three parts:

1. Evolution of Information Systems
2. Information Organization
3. Data Representations and the Data Independent Accessing Model

The first part consists of a short historical survey of the development of database systems prior to 1973. The second part describes "the entity set model," which provides a basis for describing a given enterprise in terms of entities and entity sets (it corresponds to the conceptual level of the ANSI/SPARC architecture). The third part is the most original and significant part of the paper: It forms an introduction to the *Data Independent Accessing Model* (DIAM), which is an attempt to describe a database in terms of four successive levels of abstraction,

namely the entity set (highest), string, encoding, and physical device levels. These four levels can be thought of as a more detailed, but still abstract, definition of the conceptual and internal portions of the ANSI/SPARC architecture. They can be briefly described as follows:

- *Entity set level:* Analogous to the ANSI/SPARC conceptual level.
- *String level:* Access paths to data are defined as ordered sets or "strings" of data objects. Three types of string are described—atomic strings (example: a string connecting stored field occurrences to form a part stored record occurrence), entity strings (example: a string connecting part stored record occurrences for red parts), and link strings (example: a string connecting a supplier stored record occurrence to part stored record occurrences for parts supplied by that supplier).
- *Encoding level:* Data objects and strings are mapped into linear address spaces, using a simple representation primitive known as a *basic encoding unit*.
- *Physical device level:* Linear address spaces are allocated to formatted physical subdivisions of real recording media.

The aim of DIAM, like that of references [A.11–A.12], is (in part) to provide a basis for a systematic theory of storage structures and access methods. One criticism (which applies to the formalisms of references [A.11] and [A.12] also, incidentally) is that sometimes the best method of dealing with some given access request is simply to sort the data, and sorting is of course dynamic, whereas the structures described by DIAM (and the models of [A.11] and [A.12]) are by definition always static.

A.14 S. B. Yao. "An Attribute Based Model for Database Access Cost Analysis." *ACM TODS 2*, No. 1 (March 1977).

The purpose of this paper is similar to that of references [A.11] and [A.12]; in some respects, in fact, it can be regarded as a sequel to those earlier papers, in that it presents a generalized model of database storage structures that can be seen as a combination and extension of the proposals of those papers. It also presents a set of generalized access algorithms and cost equations for that generalized model. References are given to a number of other papers that report on experiments with an implemented physical file design analyzer based on the ideas of this paper.

A.15 D. S. Batory. "Modeling the Storage Architectures of Commercial Database Systems." *ACM TODS 10*, No. 4 (December 1985).

Presents a set of primitive operations, called *elementary transformations*, by which the mapping from the conceptual schema to the corresponding internal schema (i.e., the conceptual/internal mapping—see Chapter 2) can be made explicit, and hence properly studied. The elementary transformations include *augmentation* (extending a stored record by the inclusion of prefix data as well as user data), *encoding* (converting data to an internal form by, e.g., compression), *segmentation* (splitting a record into several pieces for storage purposes), and several others. The paper claims that any conceptual/internal mapping can be represented by an appropriate sequence of such elementary transformations, and hence that the transformations could form the basis of an approach to automating the development of data management software. By way of illustration, the paper applies the ideas to the analysis of three real-world systems: INQUIRE, ADABAS, and System 2000.

A.16 R. Bayer and C. McCreight. "Organization and Maintenance of Large Ordered Indexes." *Acta Informatica 1*, No. 3 (1972).

A.17 Douglas Comer. "The Ubiquitous B-tree." *ACM Comp. Surv. 11*, No. 2 (June 1979).

A good tutorial on B-trees.

A.18 R. E. Wagner. "Indexing Design Considerations." *IBM Sys. J. 12*, No. 4 (1973).

Describes basic indexing concepts, with details of the techniques—including compression techniques—used in IBM's Virtual Storage Access Method, VSAM.

A.19 H. K. Chang. "Compressed Indexing Method." *IBM Technical Disclosure Bulletin 11*, No. 11 (April 1969).

A.20 Gopal K. Gupta. "A Self-Assessment Procedure Dealing with Binary Search Trees and B-trees." *CACM 27*, No. 5 (May 1984).

A.21 Vincent Y. Lum. "Multi-Attribute Retrieval with Combined Indexes." *CACM 13*, No. 11 (November 1970).

The paper that introduced the technique of indexing on field combinations.

A.22 James K. Mullin. "Retrieval-Update Speed Tradeoffs Using Combined Indices." *CACM 14*, No. 12 (December 1971).

A sequel to reference [A.21] that gives performance statistics for the combined index scheme for various retrieval/update ratios.

A.23 Ben Shneiderman. "Reduced Combined Indexes for Efficient Multiple Attribute Retrieval." *Information Systems 2*, No. 4 (1976).

Proposes a refinement of Lum's combined indexing technique [A.21] that considerably reduces the storage space and search time overheads. For example, the index combination *ABCD, BCDA, CDAB, DABC, ACBD, BDAC*—see the answer to Exercise A.9(b)—could be replaced by the combination *ABCD, BCD, CDA, DAB, AC, BD*. If each of *A, B, C, D* can assume 10 distinct values, then in the worst case the original combination would involve 60,000 index entries, the reduced combination only 13,200 entries.

A.24 R. Morris. "Scatter Storage Techniques." *CACM 11*, No. 1 (January 1968).

This paper is concerned primarily with hashing as it applies to the symbol table of an assembler or compiler. Its main purpose is to describe an **indirect** hashing scheme based on *scatter tables*. A scatter table is a table of record addresses, somewhat akin to the directory used in extendable hashing [A.28]. As with extendable hashing, the hash function hashes into the scatter table, not directly to the records themselves; the records themselves can be stored anywhere that seems convenient. The scatter table can thus be thought of as a single-level *index* to the underlying data, but an index that can be accessed directly via a hash instead of having to be sequentially searched. Note that a given stored data file could conceivably have several distinct scatter tables, thus in effect providing hash access to the data on several distinct hash fields (at the cost of an extra I/O for any given hash access).

Despite its programming language orientation, the paper provides a good introduction to hashing techniques in general, and most of the material is applicable to database hashing also.

A.25 W. D. Maurer and T. G. Lewis. "Hash Table Methods." *ACM Comp. Surv. 7*, No. 1 (March 1975).

A good tutorial, though now somewhat dated (it does not discuss any of the newer approaches, such as extendable hashing). The topics covered include basic hashing techniques (not just division/remainder, but also random, midsquare, radix, algebraic coding, folding, and digit analysis techniques); collision and bucket overflow handling; some theoretical analysis of the various techniques; and alternatives to hashing (techniques to be used when hashing either cannot or should not be used). *Note:* A **bucket** in hashing terminology is the unit of storage—typically a page—whose address is computed by the hash function. A bucket normally contains several records.

A.26 V. Y. Lum, P. S. T. Yuen, and M. Dodd. "Key-to-Address Transform Techniques: A Fundamental Performance Study on Large Existing Formatted Files." *CACM 14*, No. 4 (April 1971).

An investigation into the performance of several different "basic" (i.e., nonextendable) hashing algorithms. The conclusion is that the division/remainder method seems to be the best all-around performer.

A.27 M. V. Ramakrishna. "Hashing in Practice: Analysis of Hashing and Universal Hashing." Proc. 1988 ACM SIGMOD International Conference on Management of Data, Chicago, Ill. (June 1988).

As this paper points out (following Knuth [A.1]), any system that implements hashing has to solve two problems that are almost independent of one another: It has to choose among the wide variety of possible hash functions to provide one that is effective, and it also has to provide an effective technique for dealing with collisions. The author claims that, while much research has been devoted to the second of these problems, very little has been done on the first, and few attempts have been made to compare the performance of hashing in practice with the performance that is theoretically achievable (reference [A.26] is an exception). How then does a system implementer choose an appropriate hash function? This paper claims that it is possible to choose a hash function that in practice does yield performance close to that predicted by theory, and presents a set of theoretical results in support of this claim.

A.28 Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger, and H. Raymond Strong. "Extendible Hashing—A Fast Access Method for Dynamic Files." *ACM TODS 4*, No. 3 (September 1979).

A.29 G. D. Knott. "Expandable Open Addressing Hash Table Storage and Retrieval." Proc. 1971 ACM SIGFIDET Workshop on Data Description, Access, and Control, San Diego, Calif. (November 1971).

A.30 P.-Å. Larson. "Dynamic Hashing." *BIT 18* (1978).

A.31 Witold Litwin. "Virtual Hashing: A Dynamically Changing Hashing." Proc. 4th International Conference on Very Large Data Bases, Berlin, FDR (September 1978).

A.32 Witold Litwin. "Linear Hashing: A New Tool for File and Table Addressing." Proc. 6th International Conference on Very Large Data Bases, Montreal, Canada (October 1980).

A.33 Per-Åke Larson. "Linear Hashing with Overflow-Handling by Linear Probing." *ACM TODS 10*, No. 1 (March 1985).

A.34 Per-Åke Larson. "Linear Hashing with Separators—A Dynamic Hashing Scheme Achieving One-Access Retrieval." *ACM TODS 13*, No. 3 (September 1988).

References [A.28–A.34] all present extendable hashing schemes of one kind or another. The proposals of [A.29] for "expandable" hashing predate (and are therefore of course quite independent of) all of the others. Nevertheless, expandable hashing is fairly similar to extendable hashing as defined in reference [A.28], and so too is "dynamic" hashing [A.30], except that both schemes use a tree-structured directory instead of the simple contiguous directory proposed in reference [A.28]. "Virtual" hashing [A.31] is somewhat different; see the paper for details. "Linear" hashing, introduced in [A.32] and refined in [A.33] and [A.34], is an improvement on virtual hashing.

A.35 Witold Litwin. "Trie Hashing." Proc. 1981 ACM SIGMOD International Conference on Management of Data, Ann Arbor, Mich. (April 1981).

Presents an extendable hashing scheme with a number of desirable properties:

- It is *order-preserving* (that is, the "physical" sequence of stored records corresponds to the logical sequence of those records as defined by values of the hash field).
- It avoids the problems of complexity, etc., usually encountered with order-preserving hashes.
- An arbitrary record can be accessed (or shown not to exist) in a single disk access, even if the file contains many millions of records.
- The file can be arbitrarily volatile (by contrast, many hash schemes, at least of the non-extendable variety, tend to work rather poorly in the face of high insert volumes).

The hash function itself (which changes with time, as in all extendable hashing algorithms) is represented by a trie structure [A.42], which is kept in main memory whenever the file is in use and grows gracefully as the data file grows. The data file itself is, as already mentioned, kept in "physical" sequence on values of the hash field; and the logical sequence of leaf entries in the trie structure corresponds, precisely, to that "physical" sequence of the data records. Overflow

in the data file is handled via a page-splitting technique, basically like the page-splitting technique used in a B-tree.

Trie hashing looks very interesting. Like other hash schemes, it provides better performance than indexing for direct access (one I/O *vs.* typically two or three for a B-tree); and it is preferable to most other hash schemes in that it is order-preserving, which means that sequential access will also be fast. No B-tree or other additional structure is required to provide that fast sequential access. However, note the assumption that the trie will fit into main memory (probably realistic enough). If that assumption is invalid (i.e., if the data file is too large), or if the order-preserving property is not required, then linear hashing [A.32] or some other technique might provide a preferable alternative.

A.36 David B. Lomet. "Bounded Index Exponential Hashing." *ACM TODS 8*, No. 1 (March 1983).

Another extendable hashing scheme. The author claims that the techniques proposed in this paper:

- Provide direct access to any record in close to one I/O on average (and never more than two);
- Yield performance that is independent of the stored file size (by contrast, most extendable hashing schemes suffer from temporary performance degradation at the time a directory page split occurs, because typically all such pages need to be split at approximately the same time);
- Make efficient use of the available disk space (i.e., space utilization can be very good);
- Are straightforward to implement.

A.37 Anil K. Garg and C. C. Gotlieb. "Order-Preserving Key Transformations." *ACM TODS 11*, No. 2 (June 1986).

As explained in the annotation to [A.35] above, a hash (or "key transformation") function is *order-preserving* if the physical sequence of stored records corresponds to the logical sequence of those records as defined by values of the hash field. Order-preserving hashes are desirable for obvious reasons. One simple function that is clearly order-preserving is the following:

hash address = quotient after dividing hash field value
by some constant (say 10,000)

However, an obvious problem with a function such as this one is that it performs very poorly if hash field values are nonuniformly distributed (which is the usual case, of course). Hence some researchers have proposed the idea of *distribution-dependent* (but order-preserving) hash functions, i.e., functions that transform nonuniformly distributed hash field values into uniformly distributed hash addresses, while maintaining the order-preserving property. (*Note:* Trie hashing [A.35] is an example of such an approach.) The present paper gives a method for constructing such hash functions for real-world data files and demonstrates the practical feasibility of those functions.

A.38 M. V. Ramakrishna and Per-Åke Larson. "File Organization Using Composite Perfect Hashing." *ACM TODS 14*, No. 2 (June 1989).

A hash function is called *perfect* if it produces no overflows. (*Note:* "No overflows" does not mean "no collisions." For example, if we assume that the hash function generates page addresses, not record addresses—see the remark on "buckets" in the annotation to [A.25] above—and if each page can hold n stored records, then the hash function will be perfect if it never maps more than n records to the same page.) A perfect hash function has the property that any record can be retrieved in a single disk I/O. This paper presents a practical method for finding and using such perfect functions.

A.39 D. A. Huffman. "A Method for the Construction of Minimum Redundancy Codes." Proc. IRE 40 (September 1952).

A.40 B. A. Marron and P. A. D. de Maine. "Automatic Data Compression." *CACM 10*, No. 11 (November 1967).

Gives two compression/decompression algorithms: NUPAK, which operates on numeric data, and ANPAK, which operates on alphanumeric or "any" data (i.e., any string of bits).

A.41 Dennis G. Severance and Guy M. Lohman. "Differential Files: Their Application to the Maintenance of Large Databases." *ACM TODS 1*, No. 3 (September 1976).

Discusses "differential files" and their advantages. The basic idea is that updates are not made directly to the database itself, but instead are recorded in a physically distinct file—the differential file—and are merged with the actual database at some suitable subsequent time. The following advantages are claimed for such an approach:

- Database dumping costs are reduced.
- Incremental dumping is facilitated.
- Dumping and reorganization can both be performed concurrently with updating operations.
- Recovery after a program error is fast.
- Recovery after a hardware failure is fast.
- The risk of a serious data loss is reduced.
- "Memo files" are supported efficiently. (A memo file is a kind of scratchpad copy of some portion of the database, used to provide quick access to data that is probably up to date and correct but is not guaranteed to be so. See the discussion of *snapshots* in Chapter 17.)
- Software development is simplified.
- The main file software is simplified.
- Future storage costs might be reduced.

One problem not discussed is that of supporting efficient sequential access to the data—e.g., via an index—when some of the records are in the real database and some are in the differential file.

A.42 E. Fredkin. "TRIE Memory." *CACM 3*, No. 9 (September 1960).

A **trie** is a tree-structured data file (rather than a tree-structured access path to such a file; that is, the data is represented *by* the tree, it is not pointed to *from* the tree—unless the "data file" is really an index to some other file, as it effectively is in trie hashing [A.35]). Each node in a trie logically consists of n entries, where n is the number of distinct symbols available for representing data values. For example, if each data item is a decimal integer, then each node will have exactly ten entries, corresponding to the ten symbols 0, 1, 2, . . . 9. Consider the data item "4285." The (unique) node at the top of the tree will include a pointer in the "4" entry. That pointer will point to a node corresponding to all existing data items having "4" as their first digit. That node in turn (the "4 node") will include a pointer in its "2" entry to a node corresponding to all data items having "42" as their first two digits (the "42 node"). The "42" node will have a pointer in its "8" entry to the "428" node, and so on. And if (for example) there are no data items beginning "429," then the "9" entry in the "42" node will be empty (there will be no pointer); in other words, the tree is pruned to contain only nodes that are nonempty. (A trie is thus generally not a balanced tree.)

Note: The term *trie* derives from "retrieval," but is nevertheless usually pronounced *try*. Tries are also known as *radix search trees* or *digital search trees*.

A.43 Eugene Wong and T. C. Chiang. "Canonical Structure in Attribute Based File Organization." *CACM 14*, No. 9 (September 1971).

Proposes a novel storage structure based on Boolean algebra. It is assumed that all access requests are expressed as a Boolean combination of elementary "*field* = *value*" conditions, and that those elementary conditions are all known. Then the file can be partitioned into disjoint subsets for storage purposes. The subsets are the "atoms" of the Boolean algebra consisting of the set of all sets of records retrievable via the original Boolean access requests. The advantages of such an arrangement include the following:

- Set intersection of atoms is never necessary.
- An arbitrary Boolean request can easily be converted into a request for the union of one or more atoms.
- Such a union never requires the elimination of duplicates.

A.44 Michael Stonebraker. "Operating System Support for Database Management." *CACM 24*, No. 7 (July 1981).

Discusses reasons why various operating system facilities—in particular, the operating system file manager—frequently do not provide the kind of services required by the DBMS, and suggests some improvements to those facilities.

A.45 M. Schkolnick. "A Survey of Physical Database Design Methodology and Techniques." Proc. 4th International Conference on Very Large Data Bases, Berlin, FDR (September 1978).

A.46 S. Finkelstein, M. Schkolnick, and P. Tiberio. "Physical Database Design for Relational Databases." *ACM TODS 13*, No. 1 (March 1988).

In some respects, the problem of physical database design is more difficult in a relational system than it is in other kinds of system. This is because it is the system, not the user, that decides how to "navigate" through the storage structure; thus, the system will only have a chance of performing well if the storage structures chosen by the database designer are a good fit with what the system actually needs—which implies that the designer has to understand in some detail how the system works internally. And designers typically will not have such knowledge (nor is it desirable that they should). Hence some kind of automated physical design tool is highly desirable. This paper reports on such a tool, called DBDSGN, which was developed to work with System R [8.12–8.13]. DBDSGN takes as input a workload definition (i.e., a set of user requests and their corresponding execution frequencies), and produces as output a suggested physical design (i.e., a set of indexes for each stored table, typically including a "clustering index"—see the answer to Exercise A.6—for each such table). It interacts with the system optimizer (see Chapters 3 and 18) to obtain information such as the optimizer's understanding of the database (i.e., table sizes, etc.) and the cost formulas that the optimizer uses.

DBDSGN was used as the basis for an IBM product called RDT, which is a design tool for SQL/DS (see Appendix B).

A.47 Kenneth C. Sevcik. "Data Base System Performance Prediction Using an Analytical Model." Proc. 7th International Conference on Very Large Data Bases, Cannes, France (September 1981).

As its title implies, the scope of this paper is broader than that of the present appendix—it is concerned with overall system performance issues, not just with storage structures as such. The author proposes a layered framework in which various design decisions, and the interactions among those decisions, can be systematically studied. The layers of the framework represent the system at increasingly detailed levels of description; thus, each layer is more specific (i.e., at a lower level of abstraction) than the previous one. The names of the layers give some idea of the corresponding levels of detail: abstract world, logical database, physical database, data unit access, physical I/O access, and device loadings. The author claims that an analytical model based on this framework could be used to predict numerous performance characteristics, including device utilization, transaction throughput, and response times.

The paper includes an extensive bibliography on system performance (with commentary). In particular, it includes a brief survey of work on the performance of different storage structures.

A.48 Hanan Samet. "The Quadtree and Related Hierarchical Data Structures." *ACM Comp. Surv. 16*, No. 2 (June 1984).

The storage structures described in the present appendix work well for traditional commercial databases. However, as the field of database technology expands to include new kinds of data—e.g., spatial data, such as might be found in image-processing or cartographic applica-

tions—so new methods of data representation at the storage level are needed also. This paper is a tutorial introduction to some of those new methods. See Samet's book [A.49] for further discussion, also references [A.50–A.59].

A.49 Hanan Samet. *The Design and Analysis of Spatial Data Structures*. Reading, Mass.: Addison-Wesley (1990).

See the annotation to the previous reference.

A.50 Stavros Christodoulakis and Daniel Alexander Ford. "Retrieval Performance Versus Disc Space Utilization on WORM Optical Discs." Proc. 1989 ACM SIGMOD International Conference on Management of Data, Portland, Ore. (May/June 1989).

See the annotation to reference [A.51].

A.51 David Lomet and Betty Salzberg. "Access Methods for Multiversion Data." Proc. 1989 ACM SIGMOD International Conference on Management of Data, Portland, Ore. (May/June 1989).

A "WORM disk" is an optical disk with the property that once a record has been written to the disk, it can never be rewritten—i.e., it cannot be updated in place. (WORM is an acronym, standing for "Write Once, Read Many times.") WORM technology has reached the point where the use of WORM disks in database applications has become a practical proposition. Such disks have obvious advantages for certain applications, particularly those involving some kind of archival requirement. Traditional storage structures such as B-trees are not adequate for such disks, however, precisely because of the fact that rewriting is impossible. Hence (as explained in the annotation to reference [A.48], though for different reasons), new storage structures are needed. References [A.50] and [A.51] propose and analyze some such structures; reference [A.51], in particular, concerns itself with structures that are appropriate for applications in which a complete historical record is to be kept (i.e., applications in which data is only added to the database, never deleted).

A.52 J. Encarnaçao and F. L. Krause (eds.). *File Structures and Data Bases for CAD*. New York, N.Y.: North-Holland (1982).

Computer-aided design (CAD) applications are a major driving force behind the research into new storage structures. This book consists of the proceedings of a workshop on the subject, with major sections as follows:

1. Data modeling for CAD
2. Data models for geometric modeling
3. Databases for geometric modeling
4. Hardware structures
5. CAD database research issues
6. Implementation problems in CAD database systems
7. Industrial applications

A.53 R. A. Finkel and J. L. Bentley. "Quad-Trees—A Data Structure for Retrieval on Composite Keys." *Acta Informatica 4* (1974).

A.54 J. Nievergelt, H. Hinterberger, and K. C. Sevcik. "The Grid File: An Adaptable, Symmetric, Multikey File Structure." *ACM TODS 9*, No. 1 (March 1984).

A.55 Antonin Guttman. "R-Trees: A Dynamic Index Structure for Spatial Searching." Proc. ACM SIGMOD International Conference on Management of Data, Boston, Mass. (June 1984).

A.56 Nick Roussopoulos and Daniel Leifker. "Direct Spatial Search on Pictorial Databases Using Packed R-Trees." Proc. ACM SIGMOD International Conference on Management of Data, Austin, Texas (May 1985).

A.57 D. A. Beckley, M. W. Evens, and V. K. Raman. "Multikey Retrieval from K-D Trees and Quad-Trees." Proc. ACM SIGMOD International Conference on Management of Data, Austin, Texas (May 1985).

A.58 Michael Freeston. "The BANG File: A New Kind of Grid File." Proc. ACM SIGMOD International Conference on Management of Data, San Francisco, Calif. (May 1987).

A.59 Michael F. Barnsley and Alan D. Sloan. "A Better Way to Compress Images." *BYTE 13*, No. 1 (January 1988).

Describes a novel compression technique for images called *fractal compression*. The technique works by not storing the image itself at all, but rather storing code values that can be used to recreate the desired image as and when needed by means of appropriate fractal equations. In this way "compression ratios of 10,000 to 1—or even higher" can be achieved.

Answers to Selected Exercises

A.5 The advantages of indexes are as follows:

- They speed up direct access based on a given value for the indexed field or field combination. Without the index, a sequential scan would be required.
- They speed up sequential access based on the indexed field or field combination. Without the index, a sort would be required.

The disadvantages are as follows:

- They take up space on the disk. The space taken up by indexes can easily exceed that taken up by the data itself in a heavily indexed database.
- While an index will probably speed up retrieval operations, it will at the same time slow down update operations. Any INSERT or DELETE on the indexed file or UPDATE on the indexed field or field combination will require an accompanying update to the index.

A.6 In order to maintain the desired clustering, the DBMS needs to be able to determine the appropriate physical insert point for a new supplier record. This requirement is basically the same as the requirement to be able to locate a particular record given a value for the clustering field. In other words, the DBMS needs an appropriate access structure—for example, an index—based on values of the clustering field. *Note:* An index that is used in this way to help maintain physical clustering is sometimes called a **clustering index**. In general, a given stored file can have at most one clustering index, by definition.

A.7 Let the hash function be h, and suppose we wish to retrieve the record with hash field value k.

- One obvious problem is that it is not immediately clear whether the record stored at hash address $h(k)$ is the desired record or is instead a collision record that has overflowed from some earlier hash address. Of course, this question can easily be resolved by inspecting the value of the hash field in the stored record.
- Another problem is that, for any given value of $h(k)$, we need to be able to determine when to stop the process of sequentially searching for any given record. This problem can be solved by keeping an appropriate flag in the stored record prefix.
- Third, as pointed out in the introduction to the subsection on extendable hashing, when the stored file gets close to full, it is likely that most records will not be stored at their hash address location but will instead have overflowed to some other position. If record $r1$ overflows and is therefore stored at hash address $h2$, a record $r2$ that subsequently hashes to $h2$ might be forced to overflow to $h3$—even though there might as yet be no records that actually hash to $h2$ as such. In other words, the collision-handling technique itself can lead to further collisions. As a result, the average access time will go up, perhaps considerably.

A.9 (a) 3. (b) 6. For example, if the four fields are *A*, *B*, *C*, *D*, and if we use the appropriate ordered combination of field names to denote the corresponding index, the following indexes will suffice: *ABCD*, *BCDA*, *CDAB*, *DABC*, *ACBD*, *BDAC*. (c) In general, the number of indexes required is equal to the number of ways of selecting n elements from a set of N elements, where n is the smallest integer greater than or equal to $N/2$—i.e., the number is $N! / (n! * (N-n)!)$. For proof see Lum [A.21].

A.10 The number of *levels* in the B-tree is the unique positive integer k such that $n^{k-1} < N \leq n^k$. Taking logs to base n, we have $k - 1 < \log_n N \leq k$; hence

$$k = \text{ceil}(\log_n N),$$

where ceil(x) denotes the smallest integer greater than or equal to x.

Now let the number of pages in the ith level of the index be P_i (where $i = 1$ corresponds to the lowest level). We show that

$$P_i = \text{ceil}\left(\frac{N}{n^i}\right)$$

and hence that the total number of pages is

$$\sum_{i=1}^{i=k} \text{ceil}\left(\frac{N}{n^i}\right)$$

Consider the expression

$$\text{ceil}\left(\frac{\text{ceil}\left(\frac{N}{n^i}\right)}{n}\right) = x, \text{ say.}$$

Suppose $N = qn^i + r$ $(0 \leq r \leq n^i - 1)$. Then

(a) If $r = 0$,

$$x = \text{ceil}\left(\frac{q}{n}\right)$$

$$= \text{ceil}\left(\frac{qn^i}{n^{i+1}}\right)$$

$$= \text{ceil}\left(\frac{N}{n^{i+1}}\right)$$

(b) If $r > 0$,

$$x = \text{ceil}\left(\frac{q+1}{n}\right)$$

Suppose $q = q'n + r'$ $(0 \leq r' \leq n - 1)$. Then $N = (q'n + r')n^i + r = q'n^{i+1} + (r'n^i + r)$; since $0 < r \leq n^i - 1$ and $0 \leq r' \leq n - 1$,

$$0 < (r'n^i + r) \leq n^{i+1} - (n^i - n^{i+1}) < n^{i+1};$$

hence $\text{ceil}\left(\frac{N}{n^{i+1}}\right) = q' + 1$.

But

$$x = \text{ceil}\left(\frac{q'n + r' + 1}{n}\right)$$

$$= q' + 1$$

since $1 \leq r' + 1 \leq n$. Thus in both cases (a) and (b) we have that

$$\text{ceil}\left(\frac{\text{ceil}\left(\frac{N}{n^i}\right)}{n}\right) = \text{ceil}\left(\frac{N}{n^{i+1}}\right).$$

Now, it is immediate that $P_1 = \text{ceil}(N/n)$. It is also immediate that $P_{i+1} = \text{ceil}(P_i/n)$, $1 \le i < k$. Thus, if $P_i = \text{ceil}(N/n^i)$, then

$$P_{i+1} = \text{ceil}\left(\frac{\text{ceil}\left(\frac{N}{n^i}\right)}{n}\right) = \text{ceil}\left(\frac{N}{n^{i+1}}\right)$$

The rest follows by induction.

A.11

Values recorded in index	*Expanded form*
0 - 2 - Ab	Ab
1 - 3 - cke	Acke
3 - 1 - r	Ackr
1 - 7 - dams,T+	Adams,T+
7 - 1 - R	Adams,TR
5 - 1 - o	Adamso
1 - 1 - l	Al
1 - 1 - y	Ay
0 - 7 - Bailey,	Bailey,
6 - 1 - m	Baileym

Points arising:

1. The two figures preceding each recorded value represent, respectively, the number of leading characters that are the same as those in the preceding value and the number of characters actually stored.
2. The expanded form of each value shows what can be deduced from the index alone (via a sequential scan) without looking at the indexed records.
3. The "+" characters in the fourth line represent blanks.
4. We assume that the next value of the indexed field does not have "Baileym" as its first seven characters.

The percentage saving in storage space is 100 * (150 – 35) / 150 percent = 76.67%.

The index search algorithm is as follows. Let *V* be the specified value (padded with blanks if necessary to make it 15 characters long). Then:

```
found := false ;
do for each index entry in turn ;
   expand current index entry and let expanded length = N  ;
   if expanded entry = leftmost N characters of V
   then do ;
           retrieve corresponding stored record ;
           if value in that record = V
           then found := true ;
           leave loop ;
        end ;
   if expanded entry > leftmost N characters of V
   then leave loop ;
end ;
if found = false
then /* no stored record for V exists */ ;
else /* stored record for V has been found */ ;
```

For "Ackroyd,S" we get a match on the third iteration; we retrieve the corresponding record and find that it is indeed the one we want.

For "Adams,V" we get "index entry high" on the sixth iteration, so no corresponding record exists.

For "Allingham,M" we get a match on the seventh iteration; however, the record retrieved is for "Allen,S", so it is permissible to insert a new one for "Allingham,M". (We are assuming here that the indexed field values are required to be unique.) Inserting "Allingham,M" involves the following steps.

1. Finding space and storing the new record
2. Adjusting the index entry for "Allen,S" to read

```
1 - 3 - lle
```

3. Inserting an index entry between those for "Allen,S" and "Ayres,ST" to read

```
3 - 1 - i
```

Note that the preceding index entry has to be changed. In general, inserting a new entry into the index may affect the preceding entry or the following entry, or possibly neither—but never both.

APPENDIX

B DB2: A Relational Implementation

B.1 Introduction

In order to give the reader some idea of how a real relational DBMS actually works—i.e., how it implements the features of the relational model*—we devote this appendix to a brief overview of one particular relational system, namely DB2 ("IBM DATABASE 2") from IBM Corporation [B.1–B.7]. *Note:* DB2 runs in the MVS environment. IBM also provides some other relational products for its other operating environments, as follows:

- DB2/6000 for the AIX environment
- DB2/2 (originally called the OS/2 Database Manager) for the OS/2 environment
- SQL/DS for the VM and VSE environments (IBM has recently begun to refer to this product as "DB2 for VM and VSE")
- DB2/400 (originally called SQL/400) for the OS/400 environment

These products are meant to bear a strong family resemblance, of course; in this appendix, however, we will concentrate on the MVS product specifically—i.e., the name "DB2" should be understood throughout to refer to the original DB2 product specifically (where it makes any difference). Our discussions are essentially accurate as of the release level current at the time of writing, namely DB2 Version 3 Release 1. However, we do not hesitate to omit material that is irrelevant to the purpose at hand, nor to make significant simplifications in the interests of brevity.

DB2, like most other commercial relational products, supports a dialect of SQL. In particular, it permits SQL statements to be **embedded** in programs written in a variety of languages (C, COBOL, PL/I, etc.). What this means is that a program written in (say) COBOL can include SQL statements that are intermixed with the COBOL statements, as explained in Chapter 3 (and discussed in more detail in Chapter 8). In this appendix we will concentrate on this embedded use of the language, for reasons that will quickly become apparent.

---

* Or some of them, at any rate. As explained in Chapter 3, many aspects of the model are in fact not implemented in today's commercial products.

Note: Since DB2 is an SQL system and uses the SQL terms *table*, *row*, and *column* in place of the relational terms *relation*, *tuple*, and *attribute*, we will do likewise in this appendix (as indeed we have been doing throughout this book whenever we have been dealing with SQL specifically).

B.2 Storage Structure

We begin by briefly examining the way DB2 physically represents data in storage. Fig. B.1 is a schematic representation of DB2's principal storage objects and their interrelationships. The figure is meant to be interpreted as follows:

- The total collection of stored data is divided up into a number of disjoint **databases**—several user databases and several system databases, in general. (As explained in Chapter 1, there are often good practical reasons for dividing the data among several distinct databases, and DB2, like most other systems, supports this idea.) Note that the DB2 catalog is stored in one of the system databases.

  Databases (meaning *user* databases specifically) are *the unit of start and stop*, in the sense that the system operator can make a given database available or unavailable for processing via an appropriate START or STOP command.
- Each database is divided up into a number of disjoint *spaces*—several **table spaces** and several **index spaces**, in general. A "space" corresponds to what we called a

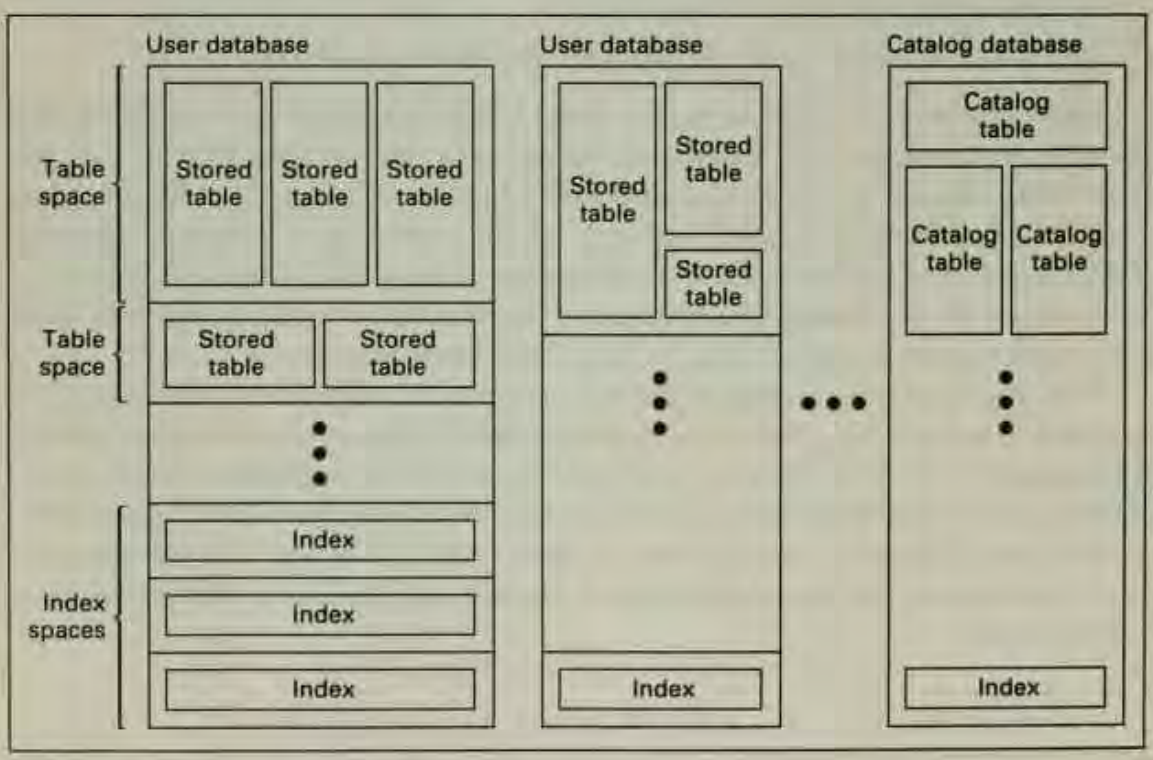

FIG. B.1 The major storage objects of DB2

page set in Appendix A; in other words, it is a dynamically extendable collection of **pages**. The pages in a given "space" are all the same size—4K bytes for index spaces, and either 4K bytes or 32K bytes for table spaces (K = 1024). At the physical level, DB2 uses the MVS access method VSAM ("Virtual Storage Access Method") to store its "spaces" on the disk.

- Each table space contains one or more **stored tables**. A stored table is the physical representation of a base table; it consists of a collection of **stored records**, one for each row of the corresponding base table.* A stored table thus corresponds to what we called a *stored file* in Appendix A (except that, for operational reasons, very large stored tables might be "partitioned" across several stored files—but this fact does not materially affect the discussions of the present appendix). A given stored table must be wholly contained within a single table space.
- Each index space contains exactly one **index** (a B-tree). A given index must be wholly contained within a single index space. A given stored table and all of its associated indexes must be wholly contained within a single database.

  Note: If a given stored table has any indexes at all, then exactly one is the **clustering index** for that stored table. A clustering index is one for which the sequence defined by the index is the same as, or close to, the physical sequence (see Exercise A.6 in Appendix A); note, therefore, that "clustering" in DB2 refers specifically to intrafile clustering, not interfile clustering. Clustering indexes are extremely important for optimization purposes—the DB2 optimizer will always try to use a clustering index, if there is one, and if the sequence represented by that index is appropriate for the user request under consideration.
- Each stored record must be wholly contained within a single page; however, one stored table can be spread over several pages, and one page can contain stored records from several stored tables. As explained above, each stored record corresponds to one row from one base table; however, a given stored record is *not* identical to the corresponding base table row. Instead, it consists of a byte string, made up of a **stored record prefix** (containing such information as the internal system identifier for the stored table of which this stored record is a part) and a set of **stored fields**. Each stored field, in turn, has (usually) a **stored field prefix** (containing such things as the data length, if the field is varying length) and an encoded form of the actual data value. *Note:* Stored data is encoded in such a manner that the IBM System/360 (/370, /390, . . .) "compare logical" instruction CLC will always yield the appropriate response when applied to two values of the same SQL data type. For example, binary integers are stored with their sign bit reversed.

  Internally, stored records are addressed by **record ID** or RID, as discussed in Appendix A.

---

* In other words, base tables in DB2 "physically exist," in the sense that they have a direct storage representation—despite the fact that, strictly speaking, physical existence is *not* the distinguishing characteristic of base tables in the relational model, as explained in Chapters 3 and 4.

B.3 System Components

The internal structure of DB2 is quite complex, as is only to be expected of a state-of-the-art system that provides all of the functions typically found in a modern DBMS. The product thus contains a very large number of internal components. From a high-level point of view, however, it can be regarded as having just four *major* components, each of which divides up into numerous subcomponents. The four major components are as follows:

1. The **system services** component, which supports system operation, operator communication, logging, and similar functions
2. The **locking services** component, which provides the necessary controls for managing concurrent access to data
3. The **database services** component, which supports the definition, retrieval, and update of user and system data
4. The **distributed data facility** component, which provides DB2's distributed database support

Of these four components, the one that is most directly relevant to the user is, of course, the database services component. We therefore concentrate here on that component specifically. That component in turn divides into five principal subcomponents, which we refer to as the *Precompiler*, *Bind*, the *Run Time Supervisor*, the *Stored Data Manager*, and the *Buffer Manager* (these terms are not always the ones used by IBM). Together, these components support (a) the preparation of application programs for execution and (b) the subsequent execution of those programs. The functions of the individual components, in outline, are as follows (refer to Fig. B.2):

- The **Precompiler** is a preprocessor for application programs that contain embedded SQL statements. It collects those statements into a **Database Request Module** (DBRM), replacing them in the original program by host language CALLs to the Run Time Supervisor.
- The **Bind** component compiles one or more related DBRMs to produce an **application plan** (i.e., executable code to implement the SQL statements in those DBRMs, including in particular calls to the Stored Data Manager). *Note:* The functions of the Bind component have been significantly extended since the DB2 product was first released. We content ourselves here with a description of the basic (original) functions only.
- The **Run Time Supervisor** oversees SQL programs during their execution. When such a program requests some database operation, control goes first to the Run Time Supervisor, thanks to the CALL inserted by the Precompiler. The Run Time Supervisor then routes control to the application plan, and the application plan in turn invokes the Stored Data Manager to perform the required function.
- The **Stored Data Manager** manages the stored database, retrieving and updating records as requested by application plans. In other words, the Stored Data Manager

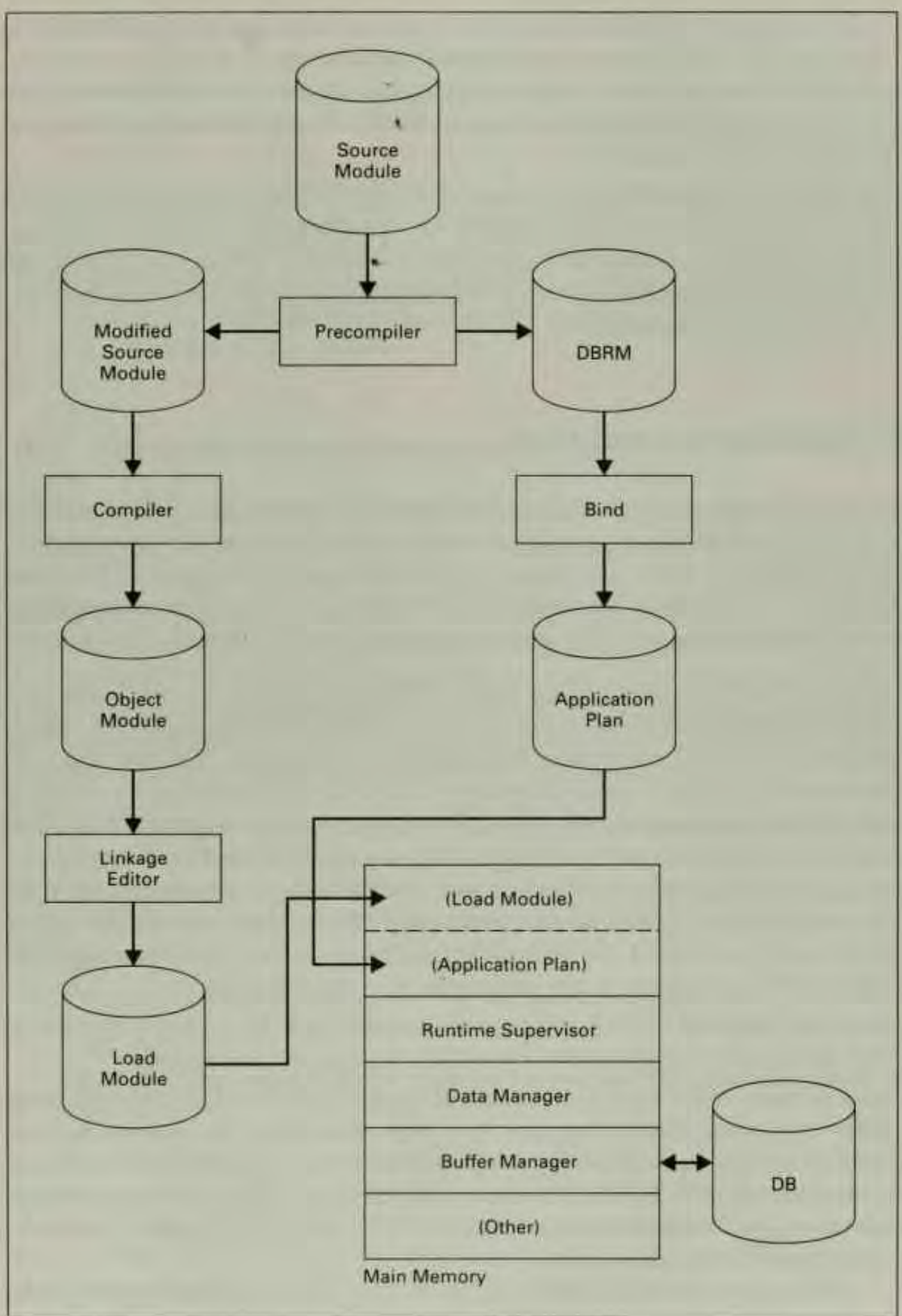

FIG. B.2 Application preparation and execution

is the component responsible for the functions attributed to the *file manager* in Appendix A of this book. It invokes other components as necessary to perform subsidiary functions such as locking, logging, sorting, etc., during the performance of its basic task; in particular, it invokes the Buffer Manager (see below) to perform physical I/O operations.

- The **Buffer Manager** is the component responsible for physically transferring data pages between the disk and main storage; i.e., it performs the actual I/O operations, as indicated above. To use the terminology of Appendix A once again, just as the Stored Data Manager corresponds to the file manager, so the Buffer Manager corresponds to the *disk manager*.

B.4 Detailed Control Flow

Let us examine the ideas of the previous section in a little more depth. Refer to Fig. B.2 once again. That figure shows the major steps involved in the preparation and execution of a DB2 application. To fix our ideas, let us assume the original program *P* is written in COBOL (we take COBOL for definiteness; the overall process is of course essentially the same for other languages). The steps that program *P* must go through are as follows:

1. Before *P* can be compiled by the regular COBOL compiler, it must first be processed by the DB2 *Precompiler*. As explained above, the Precompiler is a preprocessor for the DB2 application programming languages (C, COBOL, etc.). Its function is to analyze a source program written in one of those languages, stripping out the SQL statements it finds and replacing them by host language CALL statements. At execution time those CALLs will pass control to the Run Time Supervisor. From the SQL statements it encounters, the Precompiler constructs a Database Request Module (DBRM), which subsequently becomes input to the Bind component. *Note:* The DBRM can be regarded as just an internal representation of the original SQL statements. It does *not* consist of executable code.
2. Next, the modified COBOL program is compiled and link-edited in the normal way. Let us agree to refer to the output from this step as "load module *P*."
3. Now we come to the *Bind* step. As already suggested, Bind is really an **SQL compiler:** It converts database requests (i.e., SQL statements) into executable code. (What is more, it is an *optimizing* compiler: The output from Bind is not just code, it is *optimized* code. In DB2, in other words, the all-important optimizer component is actually a subcomponent of Bind. We shall have a litle more to say about optimization in the next section.)

   The input to Bind is a DBRM (or more likely a set of several DBRMs, if the original program involved several separately compiled procedures). The output from Bind—i.e., the compiled code, which as already mentioned is called an *application plan*—is stored away in the DB2 catalog, where it can be found when needed by the Run Time Supervisor.

4. Finally we come to execution time. Since the original program has effectively been broken into two pieces (load module and application plan), those two pieces must somehow be brought back together again at execution time. This is how it works (refer to Fig. B.2). First, the load module *P* is loaded into main memory; it starts to execute in the usual way. Sooner or later it reaches the first of the CALLs inserted by the Precompiler. Control goes to the *Run Time Supervisor*. The Run Time Supervisor then retrieves the application plan from the catalog, loads it into main memory, and passes control to it. The application plan in turn invokes the *Stored Data Manager*, which performs the necessary operations on the actual stored data (invoking the *Buffer Manager* as necessary) and passes results back to the executing application as appropriate.

B.5 Compilation and Recompilation

Our discussions so far have glossed over one extremely important point, which we now explain. First, as already indicated, DB2 is a compiling system: Database requests are *compiled* (by Bind) into internal form. By contrast, many other systems—certainly all prerelational systems, to this writer's knowledge—are *interpretive* in nature. Now, compilation is certainly advantageous from the point of view of performance; it will nearly always yield better run-time performance than will interpretation [B.10]. However, it does suffer from one significant drawback: *It is possible that decisions made by the "compiler"* (actually Bind) *at compilation time are no longer valid at execution time*. The following simple example will serve to illustrate the problem:

1. Suppose program *P* is compiled on Monday, and the optimizer decides to use an index—say index *X*—in its access strategy for *P*. Then the application plan for *P* will include explicit references to *X* by name; in other words, the generated code will be *tightly bound to* (i.e., highly dependent on) the optimizer's choice of strategy.
2. On Tuesday, some suitably authorized user issues the statement

```
DROP INDEX X ;
```

3. On Wednesday, some suitably authorized user tries to execute program *P*. What happens?

What does happen is the following. When an index is dropped, DB2 examines the catalog to see which plans are dependent on that index. Any such plans it finds it marks "invalid." When the Run Time Supervisor retrieves such a plan for execution, it sees the "invalid" marker, *and therefore invokes Bind to produce a new plan*—i.e., to choose some different access strategy and then to recompile the original DBRM (which has been kept in the catalog) in accordance with that new strategy. Assuming the recompilation is successful, the new plan then replaces the old one, and the Run Time Supervisor continues with that new plan. Thus the entire recompilation process is "transparent to the user"; the only effect that might be observed is a slight delay in the execu-

tion of certain SQL statements (possibly some change in overall program performance also, of course; the point is, however, that there should be no effect on program *logic*).

Note carefully that the automatic recompilation we are talking about here is an *SQL* recompilation, not a *COBOL* recompilation. It is not the COBOL program that is invalidated by the dropping of the index, only the application plan associated with that program.

We can now see how it is possible for programs to be independent of physical access structures—more specifically, how it is possible to create and drop (destroy) such structures without at the same time having to rewrite programs. As explained in Chapter 3, SQL statements such as SELECT and UPDATE never include any explicit mention of such structures. Instead, they simply indicate what data the user is interested in; and it is DB2's responsibility—actually Bind's responsibility—to choose a way of getting to that data, and to change to another way if the old way no longer works. We say that systems like DB2 provide a high degree of data independence (more specifically, **physical** data independence, so called to distinguish it from *logical* data independence, discussed in Chapter 17): Users and user programs are not dependent on the physical structure of the stored data—even though this is a compiling system.

Note: Data independence is easier to provide in an interpretive system, because the process of "compilation" and optimization is effectively repeated every time the program is executed. But that repetition represents a significant overhead, of course, which is precisely why DB2 chose to go the compiling route.

One further point concerning the foregoing: Our example was in terms of a dropped *index*, and perhaps that is the commonest case in practice. However, a similar sequence of events occurs when other objects (not just indexes) are dropped. Thus, for example, dropping a base table will cause all plans that refer to that table to be flagged as invalid. Of course, the automatic recompilation will work in this case only if another table has been created with the same name as the old one by the time the recompilation is done (and maybe not even then, if there are significant differences between the old table and the new one).

Let us return to the index case for a moment. Given the fact that DB2 does automatic recompilations if an existing index is dropped, the reader might be wondering whether it will also do such automatic recompilations if a new index is created. The answer is no, it will not. And the reason is that there can be no guarantee in such a case that recompiling will actually be profitable; recompilation might simply mean a lot of unnecessary work (existing compiled code might already be using an optimum strategy). The situation is different with DROP—a program will simply not work if it relies on a nonexistent object (index or otherwise), so recompiling is mandatory in this case. Thus, if some user—typically the DBA—creates a new index, and suspects that some plan could now profitably be replaced, then he or she must explicitly request that replacement by means of an explicit "REBIND" command.* Even then, of course, there is no guarantee that the new plan will in fact use the new index.

---

* It is not impossible that DB2 might be extended in some future release to perform automatic rebinds on index creation as well.

We conclude this section by noting that SQL is *always* compiled in DB2, never interpreted, even when the statements in question are submitted interactively. In other words, if the user enters (say) a SELECT statement at the terminal, then that statement will be compiled and an application plan generated for it; that plan will then be executed; and finally, after execution has completed, that plan will be discarded. Experiments have indicated that, even in the interactive case, compilation almost always results in better overall performance than interpretation [B.10]. The advantage of compilation is that the actual process of physically accessing the required data is more efficient, since it is done by tailored, compiled code. The disadvantage, of course, is that there is a cost in doing the compilation, i.e., in producing that compiled code in the first place. But the advantage almost always outweighs the disadvantage, sometimes dramatically so.

B.6 Utilities and Related Matters

In this section we take a brief look at what is involved in controlling and managing the day-to-day operation of a system like DB2. There are numerous administration and operational tasks that need to be performed, and the system provides a variety of utility programs and similar tools to assist in those tasks. We briefly describe some of the DB2 utilities in a little more detail in order to give a slightly clearer picture of the kinds of functions that need to be performed by such utilities in general. We also offer a brief description of a few miscellaneous topics of a general "utilities" nature. Please note, however, that our descriptions are very much simplified; more details can be found in reference [B.7], from which the following outline descriptions have been extracted.

- **CHECK:** The CHECK utility is used to perform certain checks on the stored data—e.g., to check that a given index is consistent with the table it indexes, or to check that a given table does not contain any "dangling" foreign key references (i.e., foreign key values for which no corresponding row exists in the target table; note that such dangling references might occur as a result of running the LOAD utility—see below—with referential integrity checking disabled).
- **COPY:** The COPY utility creates a full or incremental backup copy (an *image copy*) of a database or some portion thereof (see the discussion of MERGECOPY and RECOVER below). An incremental copy is a copy of just the pages that have been updated since the previous full or incremental copy was taken.
- **LOAD:** The LOAD utility loads data from a sequential file into one or more stored tables within a specified database. The tables in question need not be initially empty. Note that LOAD can be run with either logging or referential integrity checking disabled (or both, of course).
- **MERGECOPY:** The MERGECOPY utility merges a full copy and one or more incremental copies to produce an up-to-date full copy, or a set of incremental copies to produce an up-to-date composite incremental copy.

- **QUIESCE:** The QUIESCE utility is used to quiesce operations (temporarily) on a specified collection of base tables. The quiesced state corresponds to a single point in the log, so that the collection of tables can subsequently be recovered as a unit. In particular, QUIESCE can be useful in establishing a *point of consistency* for a collection of tables that are related via referential constraints, thus ensuring that subsequent recovery to that point will restore the data to a consistent state (see the discussion of RECOVER below).
- **RECOVER:** The RECOVER utility uses the most recent full copy, any subsequent incremental copies, and any subsequent log data to recover data after (e.g.) a media failure has occurred. RECOVER can also be used to perform *point-in-time recovery* by recovering from a specific image copy or recovering to a specific point in the log.
- **REORG:** The REORG utility reorganizes a stored database or some portion thereof to reclaim wasted space and to reestablish clustering sequence (if applicable).
- **RUNSTATS:** The RUNSTATS utility computes certain specified statistics on specified stored data (e.g., cardinality for a specified stored table), and writes those statistics to the system catalog. Bind uses such statistics in its process of optimization (see Chapter 18). They can also be useful for determining when reorganization is necessary. RUNSTATS should be executed whenever a table has been loaded, an index has been created, data has been reorganized, or generally whenever there has been a significant amount of update activity on some table. It should then be followed by an appropriate set of application plan REBINDs.

Explain

EXPLAIN is a special SQL statement that can be used to obtain information regarding the optimizer's choice of access strategy for a specified SQL query. (*Note:* EXPLAIN is not included in the SQL standard [8.1], but many SQL products support it in one form or another.) The information provided includes indexes used, details of any sorts that will be needed, and—if the specified query involves any joins—the order in which tables will be joined and the methods by which the individual joins will be performed (see Chapter 18). Such information can be useful for tuning existing applications, also for determining how projected applications will perform. Here is an example:

```
EXPLAIN PLAN FOR
        SELECT S.S#, P.P#
        FROM   S, P
        WHERE  S.CITY = P.CITY ;
```

The output from EXPLAIN is placed in a special table in the database called PLAN_TABLE. The user can then interrogate that table by means of ordinary SQL SELECT statements in order to discover, for example, whether a particular index is being used or whether creating a new index might obviate the need for a sort.

Instrumentation Facility

The DB2 **Instrumentation Facility** gathers system diagnostic information, system-wide statistics, performance information, and accounting and audit data. The information can subsequently be analyzed by the DB2 **Performance Monitor** to produce a variety of reports, including the following:

- Graphical summaries of accounting and system statistics data
- Processing time information for SQL statement, Bind, utility, and command execution
- I/O summaries
- Information on locking efficiency (number of deadlocks, number of waits, etc.)
- A detailed trace of SQL statement execution

Resource Limit Facility

The DB2 **Resource Limit Facility** is a "DB2 governor." Its principal use is to allow the DBA to limit (on an individual user basis) the amount of processor time that can be consumed during the execution of SQL data manipulation operations that are entered interactively. If the specified limit is exceeded, the offending operation is canceled.

B.7 Summary

We have presented a brief overview of DB2 in order to give the reader some idea as to what is involved in a real relational implementation. First we described the principal DB2 storage objects—**databases, table** and **index spaces** (analogous to the *page sets* of Appendix A), **stored tables** (analogous to the *stored files* of Appendix A) and **indexes** (B-trees in DB2). A stored table is the stored version of a base table; DB2—like most other relational products on the market today—thus maps base tables to physical storage in a fairly direct manner, and so does not provide as much data independence as it really should (as discussed in Chapters 3 and 4).

We then took a look at application program preparation and execution. First, the **Precompiler** removes the SQL statements from the program (replacing them by CALLs) and uses those SQL statements to build a **Database Request Module** (DBRM). The program can then be compiled and link-edited in the usual way. The DBRM(s) for a given program are processed by **Bind** to produce an **application plan**—in effect, a compiled version of the original SQL statements (Bind can be thought of as an **optimizing SQL compiler**). At run time, the CALLs inserted by the Precompiler cause control to go to the **Run Time Supervisor,** which invokes the **Stored Data Manager** (DB2's analog of the *file manager* of Appendix A) to perform the desired operations on the stored data. The Stored Data Manager in turn invokes the **Buffer Manager** (DB2's analog of the *disk manager* of Appendix A) to carry out the physical page I/O's.

We also discussed the fact that SQL requests are compiled, not interpreted, in DB2, and explained how DB2 performs **automatic recompilation** if necessitated by a change in the physical storage structure— in particular, if an existing index is dropped.

Finally, we gave a very brief overview of some of the most important DB2 **utilities** (CHECK, LOAD, COPY, etc.), together with certain related facilities such as **EXPLAIN**.

Exercises

B.1 Name the four major components of DB2.

B.2 Draw a diagram showing the overall process of program preparation and program execution in DB2.

B.3 Explain how DB2 provides physical data independence.

B.4 List the principal storage objects of DB2.

B.5 What is the maximum length in bytes of a DB2 stored record? Note that the maximum width of a base table is less than this figure; why is this?

B.6 What is a clustering index? Why can a given stored table have a maximum of one such index? What are the advantages of such an index?

B.7 The DB2 Bind component includes the optimizer as an important subcomponent. The optimizer is responsible for deciding on a strategy for implementing database requests (i.e., SQL statements); in particular, it decides when and when not to make use of indexes. How does the optimizer know what indexes exist? How does it know whether a given index is a clustering index?

B.8 Explain EXPLAIN.

References and Bibliography

References [B.1–B.5] are the principal DB2 manuals from IBM. References [B.8–B.10] are concerned with various aspects of System R, the prototype predecessor to DB2 and to IBM's other SQL products (in this regard, see also references [8.12–8.16] and [18.29–18.30]). *Note:* The reader should be aware that after several commercial releases of the product, the details of DB2's internals—particularly its compilation and optimization mechanisms—have evolved considerably and no longer display much trace of their System R origins. From a DB2 perspective, therefore, the System R references are now primarily of historical interest.

B.1 IBM Corporation. *IBM DATABASE 2 Version 3 General Information*. IBM Form No. GC26-4373.

B.2 IBM Corporation. *IBM DATABASE 2 Version 3 Administration Guide* (three volumes). IBM Form No. SC26-4374.

B.3 IBM Corporation. *IBM DATABASE 2 Version 3 SQL Reference*. IBM Form No. SC26-4380.

B.4 IBM Corporation. *IBM DATABASE 2 Version 3 Application Programming and SQL Guide*. IBM Form No. SC26-4377.

B.5 IBM Corporation. *IBM DATABASE 2 Version 3 Command and Utility Reference*. IBM Form No. SC26-4378.

B.6 Various authors. *IBM Sys. J. 23*, No. 2 (DB2 Special Issue, 1984).

A series of technical articles on DB2 as it was at first release. Topics discussed include:

- DB2 under IMS
- DB2 under TSO
- Data recovery
- DB2 design, performance, and tuning [18.30]
- DB2 buffer management
- The Query Management Facility QMF

B.7 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 DB2 and some of its companion products. Much of the present appendix is based on material from this book.

B.8 Raymond A. Lorie and Bradford W. Wade. "The Compilation of a High-Level Data Language." IBM Research Report RJ2598 (August 1979).

The DB2 compilation/recompilation scheme was pioneered by System R. This paper describes the System R mechanism in some detail, without however getting into questions of optimization (see reference [18.29] for information on this latter topic).

B.9 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 compilation in System R. 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.

B.10 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.) 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* (interactive) queries, and (b) a system like System R is capable of processing several canned transactions a second, provided appropriate indexes exist in the database.

Since this paper was first published, several relational products, including in particular DB2, have achieved performance figures that are comparable to the best figures obtained with *any* database system, relational or otherwise. In particular, transaction rates in the hundreds and even thousands of canned transactions per second have been demonstrated.

Answers to Selected Exercises

B.1 The four major components of DB2 are the system services, locking services, database services, and distributed data facility components. The database services component, in turn, divides into the Precompiler, Bind, Run Time Supervisor, Data Manager, and Buffer Manager components.

B.4 The principal DB2 storage objects are databases, table spaces, index spaces, stored tables, and indexes.

B.5 Since each stored record must be wholly contained within a single page, the maximum length of a DB2 stored record is constrained by the page size (either 4K or 32K bytes). The maximum width of a base table is less than this figure because the stored record includes certain control information—e.g., in the stored record prefix—that is concealed from the user.

B.6 A clustering index is an index for which the physical sequence of the indexed data is the same as, or very close to, the logical sequence of that same data as represented by the index entries (see Appendix A). A given stored table cannot have more than one such index because—by definition—that stored table can have only one physical sequence. The advantage of a clustering index is that sequential access via the index will access each data page once only and will thus be as fast as possible.

B.7 All such information is kept in the catalog.

APPENDIX

C Logic-Based Systems

C.1 Introduction

A significant new trend has emerged within the database research community over the past few years. That trend is toward **database systems that are based on logic**. Expressions such as *logic database, inferential DBMS, expert DBMS, deductive DBMS, knowledge base, knowledge base management system (KBMS), logic as a data model, recursive query processing*, etc., etc., are now frequently encountered in the research literature. But it is not always easy to relate such terms and the ideas they represent to familiar database terms and concepts, nor to understand the motivation underlying the research from a traditional database perspective. There is a clear need for an explanation of all of this activity in terms of conventional database ideas and principles. This appendix is an attempt to meet that need.

Our aim is thus to explain what logic-based database systems are all about from the viewpoint of someone who is familiar with database technology but not necessarily with logic *per se*. As each new idea from logic is introduced, therefore, we will explain it in conventional database terms, where possible or appropriate. Of course, we have discussed certain ideas from logic in this book already, particularly in the chapter on relational calculus (Chapter 7), which is directly based on logic. However, there is more to logic-based systems than just the relational calculus, as we will see.

The structure of this appendix is as follows. Following this preliminary background section, Section C.2 provides a brief overview of the subject, with a little history. Sections C.3 and C.4 then provide an elementary (and very much simplified) treatment of *propositional calculus* and *predicate calculus*, respectively. Next, Section C.5 introduces the so-called *proof-theoretic* view of a database, and Section C.6 builds on the ideas of that section to explain what is meant by the term *deductive DBMS*. Section C.7 then discusses some approaches to the problem of *recursive query processing*, and finally Section C.8 offers a summary and a few concluding remarks.

C.2 Overview

Research on the relationship between database theory and logic goes back to at least the late 1970s, if not earlier—see, for example, references [C.5], [C.7], and [C.13]. However, the principal stimulus for the recent considerable expansion of interest in the subject seems to have been the publication in 1984 of a landmark paper by Reiter [C.15]. In that paper, Reiter characterized the traditional perception of database systems as **model-theoretic**—by which he meant, very loosely speaking, that (a) the database is seen as a set of explicit (i.e., base) relations, each containing a set of explicit tuples, and (b) executing a query can be regarded as evaluating some specified formula (i.e., truth-valued expression) over those explicit relations and tuples. (We will explain the meaning of the term "model-theoretic" a little more precisely in Section C.5.)

Reiter then went on to argue that an alternative **proof-theoretic** view was possible, and indeed preferable in certain respects. In that alternative view—again very loosely speaking—the database is seen as a set of **axioms** ("ground" axioms, corresponding to domain values and tuples in base relations, plus certain "deductive" axioms, to be discussed), and executing a query is regarded as proving that some specified formula is a logical consequence of those axioms—in other words, proving that it is a **theorem**. (Again, we will explain the term "proof-theoretic" a little more precisely in Section C.5.)

An example is in order. Consider the following query (expressed in relational calculus) against the usual suppliers-and-parts database:

```
SPX WHERE SPX.QTY > 250
```

In the traditional—i.e., model-theoretic—interpretation, we examine the shipment tuples one by one, evaluating the formula "QTY > 250" for each one in turn; the query result then consists of just those shipment tuples for which the formula evaluates to *true*. In the proof-theoretic interpretation, by contrast, we consider the shipment tuples (plus certain other items) as *axioms* of a certain **"logical theory"**; we then apply theorem-proving techniques to determine for which possible values of the variable SPX the formula "SPX.QTY > 250" is a logical consequence of those axioms within that theory. The query result then consists of just those particular values of SPX.

Of course, this example is extremely simple—so simple, in fact, that the reader might be having difficulty in seeing what the difference between the two interpretations really is. The point is, however, that the reasoning mechanism employed in the attempted proof (in the proof-theoretic interpretation) can of course be much more sophisticated than our simple example is able to convey; indeed, it can handle certain problems that are beyond the capabilities of classical relational systems, as we will see. Furthermore, the proof-theoretic interpretation carries with it an attractive set of additional features [C.15]:

- **Representational uniformity:** It becomes possible to define a database language in which domain values, tuples in base relations, "deductive axioms," queries, and integrity constraints are all represented in essentially the same way.

- **Operational uniformity:** Proof theory provides a basis for a unified attack on a variety of apparently distinct problems, including query optimization (especially semantic optimization), integrity constraint enforcement, database design (dependency theory), program correctness proofs, and other problems.
- **Semantic modeling:** Proof theory also provides a good basis on which to define various semantic modeling extensions, such as events, type hierarchies, and certain types of entity aggregation.
- **Extended application:** Finally, proof theory also provides a basis for dealing with certain issues that classical approaches have traditionally had difficulty with—for example, *disjunctive information* (e.g., "Supplier S5 supplies either part P1 or part P2, but it is not known which").

Deductive Axioms

We offer a brief and preliminary explanation of the concept, referred to a couple of times above, of a **deductive axiom** (also known as a **rule of inference**). Basically, a deductive axiom is a rule by which, given certain facts, we are able to deduce additional facts. For example, given the facts "Anne is the mother of Betty" and "Betty is the mother of Celia," there is an obvious deductive axiom that allows us to deduce that Anne is the grandmother of Celia. To jump ahead of ourselves for a moment, therefore, we might imagine a *deductive DBMS* in which the two given facts are represented as tuples in a relation, as follows:

MOTHEROF

MOTHER	DAUGHTER
Anne	Betty
Betty	Celia

These two facts would represent **ground axioms** for the system. Let us suppose also that the deductive axiom has been formally stated to the system somehow, e.g., as follows:

```
IF   MOTHEROF       ( x, y )
AND  MOTHEROF       ( y, z )
THEN GRANDMOTHEROF  ( x, z ) ;
```

(hypothetical and simplified syntax). Now the system can apply the rule expressed in the deductive axiom to the data represented by the ground axioms (in a manner to be explained in Section C.4) to deduce the result GRANDMOTHEROF (Anne,Celia). Thus, users can ask queries such as "Who is the grandmother of Celia?" or "Who are the granddaughters of Anne?" (More precisely, they can ask "Who is Anne the grandmother of?"—see Section C.4.)

Let us now attempt to relate the foregoing ideas to traditional database concepts. In traditional terms, the deductive axiom can be thought of as a *view definition*—for example:

```
CREATE VIEW GRANDMOTHEROF AS
      ( MX.MOTHER AS GRANDMOTHER, MY.DAUGHTER AS GRANDDAUGHTER )
        WHERE MX.DAUGHTER = MY.MOTHER ;
```

(here MX and MY are tuple variables ranging over MOTHEROF). Queries such as the ones mentioned above can now be framed in terms of this view:

```
GX.GRANDMOTHER WHERE GX.GRANDDAUGHTER = 'Celia'

GX.GRANDDAUGHTER WHERE GX.GRANDMOTHER = 'Anne'
```

(GX is a tuple variable ranging over GRANDMOTHEROF).

So far, therefore, all we have really done is presented a different syntax and different interpretation for material that is already basically familiar. However, we will see in later sections that there are in fact some significant differences, not illustrated by the simple examples of the present section, between logic-based systems and more traditional DBMSs.

C.3 Propositional Calculus

In this section and the next, we present a very brief introduction to some of the basic ideas of logic. The present section considers propositional calculus and Section C.4 considers predicate calculus [C.1–C.3]. We remark immediately, however, that so far as we are concerned, propositional calculus is not all that important as an end in itself; the major aim of the present section is really to pave the way for an understanding of the next one. The aim of the two sections taken together is to provide a basis on which to build in the rest of the appendix.

The reader is assumed to be familiar with the basic concepts of Boolean algebra. For purposes of reference, we state certain laws of Boolean algebra that we will be needing later on:

- *Distributive laws:*

```
f AND ( g OR  h ) ≡ ( f AND g ) OR  ( f AND h )
f OR  ( g AND h ) ≡ ( f OR  g ) AND ( f OR  h )
```

- *De Morgan's laws:*

```
NOT ( f AND g ) ≡ NOT f OR  NOT g
NOT ( f OR  g ) ≡ NOT f AND NOT g
```

Here *f*, *g*, and *h* are arbitrary truth-valued expressions.

Now we turn to logic *per se*. Logic can be defined as **a formal method of reasoning**. Because it is formal, it can be used to perform formal tasks, such as testing the validity of an argument by examining just the structure of that argument as a sequence of steps (i.e., without paying any attention to the meaning of those steps). In particular, of course, because it is formal, it can be mechanized—i.e., it can be programmed, and thus applied by the machine.

Propositional calculus and predicate calculus are two special cases of logic in general (in fact, the former is a subset of the latter). The term "calculus," in turn, is just a general term that refers to any system of symbolic computation; in the particular cases at hand, the kind of computation involved is the computation of the truth value (*true* or *false*) of certain formulas or expressions. *Note:* Naturally we assume throughout that we are dealing with two-valued logic only.

Terms

We begin by assuming that we have some collection of objects, called **constants,** about which we can make statements of various kinds. In database parlance, these constants are the elements of the underlying domains. We define a **term** as a statement involving such constants that:

- Does not involve any quantifiers (see Section C.4), and
- Either does not involve any logical connectives (see below) or is contained in parentheses, and
- Evaluates unequivocally to either *true* or *false*.

For example, "Supplier S1 is located in London" and "Supplier S2 is located in London" and "Supplier S1 supplies part P1" are all terms (they evaluate to *true*, *false*, and *true* respectively, given our usual sample data values). By contrast, "Supplier S1 supplies some part *p*" is not a term, because it contains a quantifier ("some part *p*"). Another counterexample: "Supplier S5 will supply part P1 at some time in the future" is also not a term, because it does not evaluate unequivocally to either *true* or *false*.

Formulas

Next, we define the concept of a **formula**. Formulas of the propositional calculus—more generally, of the *predicate* calculus (see the next section)—are used in database systems to represent the conditional expression that defines the result of a query (among many other things).

```
formula
    ::=   term
        | NOT term
        | term AND formula
        | term OR formula
        | term ⇒ formula

term
    ::=   atomic-formula
        | ( formula )
```

Formulas are evaluated in accordance with the truth values of their constituent terms and the usual truth tables for the connectives. Points arising:

1. An *atomic formula* is a term that involves no connectives and is not contained in parentheses.
2. The symbol "⇒" represents the *logical implication* connective. The expression $f \Rightarrow g$ is defined to be logically equivalent to the expression NOT *f* OR *g*. *Note:* We used "IF . . . THEN . . . " for this connective in Chapter 7 and elsewhere in the body of the text.
3. We adopt the usual precedence rules for the connectives (NOT, then AND, then OR, then ⇒) in order to allow us to reduce the number of parentheses that the grammar would otherwise require.

Rules of Inference

Now we come to the **rules of inference** for the propositional calculus. Many such rules exist. Each such rule is a statement of the form

```
⊨ f ⇒ g
```

(where the symbol ⊨ can be read as **"It is always the case that"**; note that we need some such symbol in order to be able to make *metastatements*, i.e., statements about statements). Here are some examples of inference rules:

1. `⊨ ( f AND g ) ⇒ f`
2. `⊨ f ⇒ ( f OR g )`
3. `⊨ ( ( f ⇒ g ) AND ( g ⇒ h ) ) ⇒ ( f ⇒ h )`
4. `⊨ ( f AND ( f ⇒ g ) ) ⇒ g`

Note: This one is particularly important. It is called the **modus ponens** rule. Informally, it says that if *f* is *true* and *f* implies *g*, then *g* must be *true* as well. For example, given the fact that each of the following (a) and (b) is *true*—

(a) I have no money;

(b) If I have no money then I will have to wash dishes;

—then we can infer that (c) is *true* as well:

(c) I will have to wash dishes.

To continue with our inference rules:

5. `⊨ ( f ⇒ ( g ⇒ h ) ) ⇒ ( ( f AND g ) ⇒ h )`
6. `⊨ ( ( f OR g ) AND ( NOT g OR h ) ) ⇒ ( f OR h )`

Note: This is another particularly important one. It is known as the **resolution** rule. We will have more to say about it under "Proofs" below and again in Section C.4.

Proofs

We now have the necessary apparatus for dealing with formal proofs (in the context of the propositional calculus). The problem of proof is the problem of determining whether some given formula *g* (the **conclusion**) is a logical consequence of some given set of formulas *f1*, *f2*, . . . *fn* (the **premises**)*—in symbols:

```
f1, f2, ..., fn ⊢ g
```

(read as "*g* **is deducible from** *f1*, *f2*, . . . *fn*"; observe the use of another metastatement symbol, ⊢). The basic method of proceeding is known as **forward chaining**. Forward chaining consists of applying the rules of inference repeatedly to the premises, and to formulas deduced from those premises, and to formulas deduced from those formulas, etc., etc., until the conclusion is deduced; in other words, the process "chains forward"

---

* Also spelled **premisses** (singular *premiss*).

from the premises to the conclusion. However, there are several variations on this basic theme:

1. **Adopting a premise:** If g is of the form $p \Rightarrow q$, adopt p as an additional premise and show that q is deducible from the given premises plus p.
2. **Backward chaining:** Instead of trying to prove $p \Rightarrow q$, prove the **contrapositive** NOT $q \Rightarrow$ NOT p.
3. **Reductio ad absurdum:** Instead of trying to prove $p \Rightarrow q$ directly, assume that p and NOT q are both *true* and derive a contradiction.
4. **Resolution:** This method uses the resolution inference rule (No. 6 in the list shown earlier).

We discuss the resolution technique in some detail, since it is of wide applicability (in particular, it generalizes to the case of predicate calculus also, as we will see in Section C.4).

Note first that the resolution rule is effectively a rule that allows us to **cancel subformulas;** that is, given the two formulas

```
f OR g      and      NOT g OR h
```

we can derive the simplified formula

```
f OR h
```

In particular, given *f* OR *g* and NOT *g* (i.e., taking *h* as *true*), we can derive *f*.

Observe, therefore, that the rule applies in general to a *conjunction* (AND) of two formulas, each of which is a *disjunction* (OR) of two formulas. In order to apply the resolution rule, therefore, we proceed as follows. (To make our discussion a little more concrete, we explain the process in terms of a specific example.) Suppose we wish to determine whether the following putative proof is in fact valid:

```
A ⇒ ( B ⇒ C ), NOT D OR A, B ⊢ D ⇒ C
```

(where *A*, *B*, *C*, and *D* are formulas). We start by adopting the negation of the conclusion as an additional premise, and then writing each premise on a separate line, as follows:

```
A ⇒ ( B ⇒ C )
NOT D OR A
B
NOT ( D ⇒ C )
```

Note that these four lines are implicitly all "ANDed" together. We now convert each individual line to **conjunctive normal form,** i.e., a form consisting of one or more formulas all ANDed together, each individual formula containing (possibly) NOTs and ORs but no ANDs (see Chapter 18). Of course, the second and third lines are already in this form. In order to convert the other two lines, we first eliminate all appearances of "⇒" (using the definition of that connective in terms of NOT and OR); we then apply the distributive laws and De Morgan's laws as necessary (see the beginning of this

section). We also drop redundant parentheses and pairs of adjacent NOTs (which cancel out). The four lines become

```
NOT A OR NOT B OR C
NOT D OR A
B
D AND NOT C
```

Next, any line that includes any explicit ANDs we replace by a set of separate lines, one for each of the individual formulas ANDed together (dropping the ANDs in the process). In the example, this step applies to the fourth line only. The premises now look like this:

```
NOT A OR NOT B OR C
NOT D OR A
B
D
NOT C
```

Now we can start to apply the resolution rule. We choose a pair of lines that can be "resolved," i.e., a pair that contain (respectively) some particular formula and the negation of that formula. Let us choose the first two lines, which contain NOT *A* and *A* respectively, and resolve them, giving

```
NOT D OR NOT B OR C
B
D
NOT C
```

(*Note:* We also need to keep the two original lines, in general, but in this particular example they will not be needed any more.) Now we apply the rule again, again choosing the first two lines (resolving NOT *B* and *B*), giving

```
NOT D OR C
D
NOT C
```

We choose the first two lines again (NOT *D* and *D*):

```
C
NOT C
```

And once again (*C* and NOT *C*); the final result is the empty proposition (usually represented thus: []), which represents a contradiction. By reductio ad absurdum, therefore, the desired result is proved.

C.4 Predicate Calculus

We now turn our attention to the predicate calculus. The big difference between propositional calculus and predicate calculus is that the latter allows formulas to contain quantifiers, which makes it very much more powerful and of very much wider applicability. For example, the statement "Supplier S1 supplies some part *p*" is not a legal

formula in propositional calculus, but it is a legal formula in predicate calculus. Hence predicate calculus provides us with a basis for expressing queries such as "What parts are supplied by supplier S1?" or "Find suppliers who supply some part" or even "Find suppliers who do not supply any parts at all."

Predicates

A **predicate** is a *truth-valued function*, i.e., a function that, given appropriate arguments, returns either *true* or *false*. For example, ">" is a predicate; the expression ">(*x*,*y*)"—more conventionally written "*x* > *y*"—returns *true* if the value of *x* is greater than the value of *y* and *false* otherwise. A predicate that takes *n* arguments is called an *n*-place predicate. A proposition—or for that matter a formula—in the sense of Section C.3 can be regarded as a zero-place predicate: It has no arguments and evaluates to either *true* or *false*, unconditionally.

It is convenient to assume that the predicates "=", ">", "≥", etc., are builtin (i.e., they are part of the formal system we are defining) and that expressions using them can be written in the conventional manner, but of course users should be able to define their own additional predicates as well. Indeed, that is the whole point, as we will quickly see: The fact is, in database terms, a user-defined predicate is nothing more nor less than a user-defined *relation*. The suppliers relation S, for example, can be regarded as a predicate with four arguments (S#, SNAME, STATUS, and CITY). Furthermore, the expressions S(S1,Smith,20,London) and S(S6,White,45,Rome) represent "instances" or "invocations" of that predicate that evaluate to *true* and *false* respectively (within the particular database that is represented by our usual sample set of values). Informally, we can regard such predicates—together with any applicable integrity constraints, which are also predicates—as defining what the corresponding relation "means," as explained in Part II of this book.

Well-Formed Formulas

The next step is to extend the definition of "formula." In order to avoid confusion with the formulas of the previous section (which are actually a special case), we will now switch to the term **well-formed formula** (WFF, pronounced "weff") that we first introduced in Chapter 7. Here is a simplified syntax for WFFs:

```
wff   ::=   term
          | NOT ( wff )
          | ( wff ) AND ( wff )
          | ( wff ) OR ( wff )
          | ( wff ) ⇒ ( wff )
          | EXISTS variable ( wff )
          | FORALL variable ( wff )

term  ::=   [ NOT ] predicate [ ( argument-commalist ) ]
```

Points arising:

1. A *term* is simply a possibly negated predicate instance. Each *argument* must be a constant, a variable, or a function reference, where each argument to a function

reference in turn is a constant or variable or function reference. The *argument commalist* (and enclosing parentheses) are omitted for a zero-place predicate. *Note:* Functions are permitted in order to allow WFFs to include computational expressions such as "+(*x*,*y*)"—more conventionally written "*x* + *y*"—and so forth.

2. As in Section C.3, we adopt the usual precedence rules for the connectives (NOT, then AND, then OR, then ⇒) in order to reduce the number of parentheses that the grammar would otherwise require.
3. The reader is assumed to be familiar with the quantifiers EXISTS and FORALL.*
4. De Morgan's laws can be generalized to apply to quantified WFFs, as follows:

```
NOT ( FORALL x ( f ) )  ≡  EXISTS x ( NOT ( f ) )
NOT ( EXISTS x ( f ) )  ≡  FORALL x ( NOT ( f ) )
```

   This point was also discussed in Chapter 7.
5. To repeat yet another point from Chapter 7: Within a given WFF, each occurrence of a variable is either **free** or **bound**. An occurrence of a variable is bound if (a) it is the variable immediately following a quantifier (the "quantified variable") or (b) it is within the scope of a quantifier and has the same name as the applicable quantified variable. A variable occurrence is free if it is not bound.
6. A **closed WFF** is one that contains no free variable occurrences. An **open WFF** is a WFF that is not closed.

Interpretations and Models

What do WFFs *mean?* In order to provide a formal answer to this question, we introduce the notion of an **interpretation**. An interpretation of a WFF—or more generally of a set of WFFs—is defined as follows:

- First, we specify a **universe of discourse** over which the WFFs are to be interpreted. In other words, we specify a *mapping* between the permitted constants of the formal system (the domain values, in database terms) and objects in "the real world." Each individual constant corresponds to precisely one element in the universe of discourse.
- Second, we specify a meaning for each predicate in terms of objects in the universe of discourse.
- Third, we also specify a meaning for each function in terms of objects in the universe of discourse.

Then the interpretation consists of the combination of the universe of discourse, plus the mapping of individual constants to objects in that universe, plus the defined meanings for the predicates and functions with respect to that universe.

By way of example, let the universe of discourse be the set of integers

---

* We are concerned here only with the *first order* predicate calculus, which basically means that there are no predicate variables (i.e., variables whose value is a predicate), and hence that predicates cannot themselves be subjected to quantification. See Exercise 7.9 in Chapter 7.

{0,1,2,3,4,5}, let constants such as "2" correspond to elements of that universe in the obvious way, and let the predicate ">" be defined to have the usual meaning. (We could also define functions such as "+", "-", etc., if desired.) Now we can assign a truth value to WFFs such as the following, as indicated:

```
2 > 1                    — true
2 > 3                    — false
EXISTS x ( x > 2 )       — true
FORALL x ( x > 2 )       — false
```

Note, however, that other interpretations are possible. For example, we might specify the universe of discourse to be a set of security classification levels, as follows:

```
destroy before reading (level 5)
destroy after reading  (level 4)
top secret             (level 3)
secret                 (level 2)
confidential           (level 1)
unclassified           (level 0)
```

The predicate ">" could now mean "more secure (i.e., higher classification) than."

Now, the reader will probably realize that the two possible interpretations just given are *isomorphic*—that is, it is possible to set up a one-to-one correspondence between them, and hence at a deep level the two interpretations are really one and the same. But it must be clearly understood that interpretations can exist that are genuinely different in kind. For example, we might once again take the universe of discourse to be the integers 0–5, but define the predicate ">" to mean *equality*. (Of course, we would probably cause a lot of confusion that way, but at least we would be within our rights to do so.) Now the first WFF above would evaluate to *false* instead of *true*.

Another point that must be clearly understood is that two interpretations might be genuinely different in the foregoing sense and yet give the same truth values for the given set of WFFs. This would be the case with the two different definitions of ">" in our example if the WFF "2 > 1" were omitted.

Note, incidentally, that all of the WFFs we have been discussing in this subsection so far have been *closed* WFFs. The reason is that, given an interpretation, it is always possible to assign a truth value unambiguously to a closed WFF, but the truth value of an open WFF will depend on the values assigned to the free variables. For example, the open WFF

```
x > 3
```

is (obviously) *true* if the value of *x* is greater than 3 and *false* otherwise (whatever "greater than" and "3" mean in the interpretation).

Now we define a **model** of a WFF, or more generally of a set of (closed) WFFs, to be an interpretation for which all WFFs in the set are *true*. The two interpretations given above for the four WFFs

```
2 > 1
2 > 3
EXISTS x ( x > 2 )
FORALL x ( x > 2 )
```

in terms of the integers 0-5 were not models for those WFFs, because some of the WFFs evaluated to *false* under that interpretation. By contrast, the first interpretation (in which ">" was defined "properly") *would* have been a model for the set of WFFs

```
2 > 1
3 > 2
EXISTS x ( x > 2 )
FORALL x ( x > 2 OR NOT ( x > 2 ) )
```

Note finally that, since a given set of WFFs can admit of several interpretations in which all of the WFFs evaluate to *true*, it can therefore have several *models* (in general). Thus, a database can have several models (in general), since—in the model-theoretic view—a database *is* basically a set of WFFs. See Section C.5.

Clausal Form

Just as any propositional calculus formula can be converted to conjunctive normal form, so any predicate calculus WFF can be converted to **clausal form,** which can be regarded as an extended version of conjunctive normal form. One motivation for making such a conversion is that (again) it allows us to apply the resolution rule in constructing or verifying proofs, as we will see.

The conversion process proceeds as follows (in outline; for more details, see reference [C.10]). We illustrate the steps by applying them to a sample WFF, namely

```
FORALL x ( p ( x ) AND EXISTS y ( FORALL z ( q ( y, z ) ) ) )
```

Here p and q are predicates and x, y, and z are variables.

1. Eliminate "⇒" symbols as in Section C.3. In our example, this first transformation has no effect.
2. Use De Morgan's laws, plus the fact that two adjacent NOTs cancel out, to move NOTs so that they apply only to terms, not to general WFFs. (Again this particular transformation has no effect in our particular example.)
3. Convert the WFF to **prenex normal form** by moving all quantifiers to the front (systematically renaming variables if necessary):

   ```
   FORALL x ( EXISTS y ( FORALL z ( p ( x ) AND q ( y, z ) ) ) )
   ```

4. Note that an existentially quantified WFF such as

   ```
   EXISTS v ( r ( v ) )
   ```

   is equivalent to the WFF

   ```
   r ( a )
   ```

   for some (unknown) constant a; that is, the original WFF asserts that some such a does exist, we just don't know its value. Likewise, a WFF such as

   ```
   FORALL u ( EXISTS v ( s ( u, v ) ) )
   ```

   is equivalent to the WFF

   ```
   FORALL u ( s ( u, f ( u ) ) )
   ```

for some (unknown) function *f* of the universally quantified variable *u*. The constant *a* and the function *f* in these examples are known, respectively, as a **Skolem constant** and a **Skolem function*** (where a Skolem constant is really just a Skolem function with no arguments). So the next step is to eliminate existential quantifiers by replacing the corresponding quantified variables by (arbitrary) Skolem functions of all universally quantified variables that precede the quantifier in question in the WFF:

```
FORALL x ( FORALL z ( p ( x ) AND q ( f ( x ), z ) ) )
```

5. All variables are now universally quantified. We can therefore adopt a convention by which all variables are *implicitly* universally quantified and so drop the explicit quantifiers:

```
p ( x ) AND q ( f ( x ), z )
```

6. Convert the WFF to conjunctive normal form, i.e., to a set of clauses all ANDed together, each clause involving possibly NOTs and ORs but no ANDs. In our example, the WFF is already in this form.
7. Write each clause on a separate line and drop the ANDs:

```
p ( x )
q ( f ( x ), z )
```

This is the clausal form equivalent of the original WFF.

Note: It follows from the foregoing procedure that the general form of a WFF in clausal form is a set of clauses, each on a line of its own, and each of the form

```
NOT A1 OR NOT A2 OR ... OR NOT Am OR B1 OR B2 OR ... OR Bn
```

where the *A*'s and *B*'s are all nonnegated terms. We can rewrite such a clause, if we like, as

```
A1 AND A2 AND ... AND Am  ⇒  B1 OR B2 OR ... OR Bn
```

If there is at most one *B* ($n = 0$ or 1), the clause is called a **Horn clause**.†

Using the Resolution Rule

Now we are in a position to see how a logic-based database system can deal with queries. We use the example from the end of Section C.2. First, we have a predicate MOTHEROF, which takes two arguments, representing mother and daughter respectively, and we are given the following two terms (predicate instances):

1. `MOTHEROF ( Anne, Betty )`
2. `MOTHEROF ( Betty, Celia )`

We are also given the following WFF (the "deductive axiom"):

3. `MOTHEROF ( x, y ) AND MOTHEROF ( y, z ) ⇒ GRANDMOTHEROF ( x, z )`

---

* After the logician T. A. Skolem.
† After the logician Alfred Horn.

(note that this is a Horn clause). In order to simplify the application of the resolution rule, let us rewrite the clause to eliminate the "⇒" symbol:

```
4. NOT MOTHEROF ( x, y ) OR NOT MOTHEROF ( y, z ) OR GRANDMOTHEROF ( x, z )
```

We now show how to prove that Anne is the grandmother of Celia—i.e., how to answer the query "Is Anne Celia's grandmother?" We begin by negating the conclusion that is to be proved and adopting it as an additional premise:

```
5. NOT GRANDMOTHEROF ( Anne, Celia )
```

Now, to apply the resolution rule, we must systematically substitute values for variables in such a way that we can find two clauses that contain, respectively, a WFF and its negation. Such substitution is legitimate because the variables are all implicitly universally quantified, and hence individual (nonnegated) WFFs must be *true* for each and every legal combinations of values of their variables. *Note:* The process of finding a set of substitutions that make two clauses resolvable in this manner is known as **unification**.

To see how the foregoing works in the case at hand, note first that lines 4 and 5 contain the terms GRANDMOTHEROF(*x*,*z*) and NOT GRANDMOTHEROF(Anne, Celia), respectively. So we substitute Anne for *x* and Celia for *z* in line 4 and resolve, to obtain

```
6. NOT MOTHEROF ( Anne, y ) OR NOT MOTHEROF ( y, Celia )
```

Line 2 contains MOTHEROF(Betty,Celia). So we substitute Betty for *y* in line 6 and resolve, to obtain

```
7. NOT MOTHEROF ( Anne, Betty )
```

Resolving line 7 and line 1, we obtain the empty clause []: Contradiction. Hence the answer to the original query is "Yes, Anne is Celia's grandmother."

What about the query "Who are the granddaughters of Anne?" Observe first of all that the system does not know about granddaughters, it only knows about grandmothers. We could add another deductive axiom to the effect that *z* is the granddaughter of *x* if and only if *x* is the grandmother of *z* (no males are allowed in this database). Alternatively, of course, we could rephrase the question as "Who is Anne the grandmother of?" Let us consider this latter formulation. The premises are (to repeat)

```
1. MOTHEROF ( Anne,  Betty )
2. MOTHEROF ( Betty, Celia )
3. NOT MOTHEROF ( x, y ) OR NOT MOTHEROF ( y, z ) OR GRANDMOTHEROF ( x, z )
```

We introduce a fourth premise, as follows:

```
4. NOT GRANDMOTHEROF ( Anne, r ) OR RESULT ( r )
```

Intuitively, this says that either Anne is not the grandmother of anyone, or alternatively

she is the grandmother of some person *r*. We wish to discover the identity of all such persons *r*, assuming they exist. We proceed as follows.

First, substitute Anne for *x* and *r* for *z* and resolve lines 4 and 3, to obtain

5. `NOT MOTHEROF ( Anne, y ) OR NOT MOTHEROF ( y, z ) OR RESULT ( z )`

Next, substitute Betty for *y* and resolve lines 5 and 1, to obtain

6. `NOT MOTHEROF ( Betty, z ) OR RESULT ( z )`

Now substitute Celia for *z* and resolve lines 6 and 2, to obtain

7. `RESULT ( Celia )`

Hence Anne is the grandmother of Celia.

Note: If we had been given an additional term, as follows—

```
MOTHEROF ( Betty, Delia )
```

—then we could have substituted Delia for *z* in the final step (instead of Celia) and obtained

```
RESULT ( Delia )
```

The user expects to see both names in the result, of course. Thus, the system needs to apply the unification and resolution process exhaustively to generate *all possible* result values. Details of this refinement are beyond the scope of the present discussion.

C.5 A Proof-Theoretic View of Databases

As explained in Section C.4, a *clause* is an expression of the form

```
A1 AND A2 AND ... AND Am ⇒ B1 OR B2 OR ... OR Bn
```

where the *A*'s and *B*'s are all terms of the form

```
r ( x1, x2, ... xt )
```

(here *r* is a predicate and *x1*, *x2*, . . . *xt* are the arguments to that predicate). Following reference [C.12], we now consider a couple of important special cases of this general construct.

1. `Case 1: m = 0, n = 1`

In this case the clause can be simplified to just

```
⇒ B1
```

or in other words (dropping the implication symbol) to just

```
r ( x1, x2, ..., xt )
```

for some predicate *r* and some set of arguments *x1*, *x2*, . . . , *xt*. If the *x*'s are all constants, the clause represents a **ground axiom**—i.e., it is a statement that is unequivo-

cally *true*. In database terms, such a statement corresponds to a tuple of some relation R^*. The predicate r corresponds to the "meaning" of relation R, as explained elsewhere in this book. For example, in the suppliers-and-parts database, there is a relation called SP, the meaning of which is that the indicated supplier (S#) is supplying the indicated part (P#) in the indicated quantity (QTY). Note that this meaning corresponds to an **open WFF,** since it contains some free variables (S#, P#, and QTY). By contrast, the tuple (S1,P1,300)—in which the arguments are all constants—is a ground axiom or **closed WFF** that asserts unequivocally that supplier S1 is supplying part P1 in a quantity of 300.

2. `Case 2: m > 0, n = 1`

In this case the clause takes the form

```
A1 AND A2 AND . . . AND Am ⇒ B
```

which can be regarded as a **deductive axiom;** it gives a (partial) definition of the predicate on the right-hand side of the implication symbol in terms of those on the left (see the definition of the GRANDMOTHEROF predicate earlier for an example).

Alternatively, such a clause might be regarded as defining an **integrity constraint**. Suppose for the sake of the example that the suppliers relation S contains only two attributes, S# and CITY. Then the clause

```
S ( s, c1 ) AND S ( s, c2 ) ⇒ c1 = c2
```

expresses the constraint that CITY is functionally dependent on S#. Note the use of the builtin predicate "=" in this example.

As the foregoing discussions demonstrate, tuples in relations ("ground axioms"), derived relations ("deductive axioms"), and integrity constraints can all be regarded as special cases of the general *clause* construct. Let us now try to see how these ideas can lead to the "proof-theoretic" view of a database mentioned in Section C.2.

First, the traditional view of a database can be regarded as **model**-theoretic. By "traditional view" here, we simply mean a view in which the database is perceived as a collection of explicitly named (base) relations, each containing a set of explicit tuples, together with an explicit set of integrity constraints that those tuples are not allowed to violate. It is this perception that can be characterized as "model-theoretic," as we now explain.

- The underlying domains contain values or constants that are supposed to stand for certain objects in the "real world" (more precisely, in some **interpretation,** in the sense of Section C.4). They thus correspond to the universe of discourse.
- The base relations (more precisely, the base relation *headings*) represent a set of predicates or open WFFs that are to be interpreted over that universe. For example, the heading of relation SP represents the predicate "Supplier S# supplies part P# in quantity QTY."

---

* Or to an element of a domain.

- Each tuple in a given relation represents an instance of the corresponding predicate; i.e., it represents a proposition (a closed WFF—it contains no variables) that is unequivocally *true* in the universe of discourse.
- The integrity constraints are also closed WFFs, and they are interpreted over the same universe. Since the data does not (i.e., *must* not!) violate the constraints, these constraints necessarily evaluate to *true* as well.
- The tuples and the integrity constraints can together be regarded as the set of axioms defining a certain **logical theory** (loosely speaking, a "theory" in logic *is* a set of axioms). Since those axioms are all *true* in the interpretation, then by definition that interpretation is a **model** of that logical theory, in the sense of Section C.4. Note that, as pointed out in that section, the model might not be unique—that is, a given database might have several possible interpretations, all of which are equally valid from a logical standpoint.

In the model-theoretic view, therefore, the "meaning" of the database *is* the model, in the foregoing sense of the term "model." And since there are many possible models, there are many possible meanings, at least in principle.* Furthermore, query processing in the model-theoretic view is essentially a process of evaluating a certain open WFF to discover which values of the free variables in that WFF cause the WFF to evaluate to *true* within the model.

So much for the model-theoretic view. However, in order to be able to apply the rules of inference described in Sections C.3 and C.4, it becomes necessary to adopt a different perspective, one in which the database is explicitly regarded as a certain logical theory, i.e., as a set of axioms. The "meaning" of the database then becomes, precisely, the collection of all *true* statements that can be deduced from the axioms using those axioms in all possible combinations—i.e., it is the set of **theorems** that can be proved from those axioms. This is the **proof-theoretic** view. In this view, query evaluation becomes a theorem-proving process (conceptually speaking, at any rate; in the interests of efficiency, however, the system is likely to use more conventional query processing techniques, as we will see in Section C.7).

Note: It follows from the foregoing paragraph that one difference between the model-theoretic and proof-theoretic views (intuitively speaking) is that, whereas a database can have many "meanings" in the model-theoretic view, it typically has precisely one "meaning" in the proof-theoretic view—except that (a) as pointed out earlier, that one meaning is really *the* canonical meaning in the model-theoretic case, and in any case (b) the remark to the effect that there is only one meaning in the proof-theoretic case ceases to be true, in general, if the database includes any negative axioms [C.9-C.10].

---

* However, if we assume that the database does not explicitly contain any negative information (e.g., a proposition of the form "NOT S#(S9)," meaning that S9 is not a supplier number), there will also be a "minimal" or *canonical* meaning, which is the intersection of all possible models [C.10]. In this case, moreover, that canonical meaning will be the same as the meaning ascribed to the database under the proof-theoretic view, to be explained.

The axioms for a given database (proof-theoretic view) can be informally summarized as follows [C.15]:

1. Ground axioms, corresponding to the elements of the domains and the tuples of the base relations. These axioms constitute what is sometimes called the **extensional database** (EDB).
2. A "completion axiom" for each relation, which states that no tuples of that relation exist other than those explicitly appearing in that relation. These axioms correspond to what is usually called the **Closed World Assumption** (CWA), which states that omission of a given tuple from a given relation implies that the proposition corresponding to that tuple is *false*. For example, the fact that the suppliers relation S does not include the tuple (S6,White,45,Rome) means that the proposition "There exists a supplier S6 named White with status 45 located in Rome" is *false*.
3. The "unique name" axiom, which states that every constant is distinguishable from all the rest (i.e., has a unique name).
4. The "domain closure" axiom, which states that no constants exist other than those in the database domains.
5. A set of axioms (essentially standard) to define the builtin predicate "=". These axioms are needed because the axioms in Nos. 2, 3, and 4 above each make use of the equality predicate.

We conclude this section with a brief summary of the principal differences between the two perceptions (model-theoretic and proof-theoretic). First of all, it has to be said that from a purely pragmatic standpoint there might not be very much difference at all!—at least in terms of present-day systems. However:

- Nos. 2–5 in the list of axioms for the proof-theoretic view make explicit certain assumptions that are implicit in the notion of interpretation in the model-theoretic view [C.15]. Stating assumptions explicitly is generally a good idea; furthermore, it is necessary to specify those additional axioms explicitly in order to be able to apply general proof techniques, such as the resolution method described in Sections C.3 and C.4.
- Note that the list of axioms makes no mention of integrity constraints. The reason for that omission is that (in the proof-theoretic view) adding such constraints converts the system into a **deductive** DBMS. See Section C.6.
- The proof-theoretic view does enjoy a certain elegance that the model-theoretic view does not, inasmuch as it provides a uniform perception of several constructs that are usually thought of as more or less distinct: base data, queries, integrity constraints (the previous point notwithstanding), virtual data (etc.). As a consequence, the possibility arises of more uniform interfaces and more uniform implementations.
- The proof-theoretic view also provides a natural basis for treating certain problems that relational systems have traditionally always had difficulty with—**disjunctive information** (e.g., "Supplier S6 is located in either Paris or Rome"), the derivation

of **negative information** (e.g., "Who is not a supplier?"), and **recursive queries** (see the next section)—though in this last case, at least, there is no reason in principle why a classical relational system could not be extended appropriately to deal with such queries (a proposal for doing so can be found in reference [C.14]). We will have more to say regarding such matters in Sections C.6 and C.7.

- Finally, to quote Reiter [C.15], the proof-theoretic view "provides a correct treatment of [extensions to] the relational model to incorporate more real world semantics" (including—to repeat from Section C.2—events, type hierarchies, and certain kinds of entity aggregation).

C.6 Deductive Database Systems

A **deductive DBMS** is a DBMS that supports the proof-theoretic view of a database, and in particular is capable of deducing additional facts from the extensional database by applying specified **deductive axioms** or **rules of inference** to those facts. The deductive axioms, together with the integrity constraints (discussed below), form what is sometimes called the **intensional database** (IDB), and the extensional database and the intensional database together constitute what is usually called the *deductive database* (not a very good term, since it is the DBMS, not the database, that carries out the deductions).

As just indicated, the deductive axioms form one part of the intensional database. The other part consists of additional axioms that represent integrity constraints (i.e., rules whose primary purpose is to constrain updates, though actually such rules can also be used in the deduction process to generate new facts).

Let us see what the suppliers-and-parts database would look like in "deductive DBMS" form. First, there will be a set of ground axioms defining the legal domain values. *Note:* In what follows, we omit the (necessary) quotes surrounding string constants purely for reasons of readability.

```
S# ( S1 )   NAME ( Smith )   STATUS (  5 )   CITY ( London )
S# ( S2 )   NAME ( Jones )   STATUS ( 10 )   CITY ( Paris  )
S# ( S3 )   NAME ( Blake )   STATUS ( 15 )   CITY ( Rome   )
S# ( S4 )   NAME ( Clark )   etc.           CITY ( Athens )
S# ( S5 )   NAME ( Adams )                   etc.
S# ( S6 )   NAME ( White )
S# ( S7 )   NAME ( Nut   )
etc.        NAME ( Bolt  )
            NAME ( Screw )
            etc.

etc., etc., etc.
```

Next, there will be ground axioms for the tuples in the base relations:

```
S ( S1, Smith, 20, London )
S ( S2, Jones, 10, Paris )
etc.
```

```
P ( P1, Nut, Red, 12, London )
etc.

SP ( S1, P1, 300 )
etc.
```

Note: We are not seriously suggesting that the extensional database will be created by explicitly listing all of the ground axioms as indicated above; rather, of course, traditional data defnition and data entry methods will be used. In other words, deductive DBMSs will typically apply their deductions to conventional databases that already exist and have been constructed in the conventional manner. Note, however, that it now becomes more important than ever that the extensional database not violate any of the declared integrity constraints!—because a database that does violate any such constraints represents (in logical terms) an inconsistent set of axioms, and it is well known that *absolutely any proposition whatsoever* can be proved to be "true" from such a starting point (in other words, contradictions can be derived). For exactly the same reason, it is also important that the stated set of integrity constraints be consistent.

Now for the intensional database. Here are the domain constraints:

```
S ( s, sn, st, sc ) ⇒ S# ( s ) AND
                      NAME ( sn ) AND
                      STATUS ( st ) AND
                      CITY ( sc )

P ( p, pn, pl, pw, pc ) ⇒ P# ( p ) AND
                           NAME ( pn ) AND
                           COLOR ( pl ) AND
                           WEIGHT ( pw ) AND
                           CITY ( pc )
etc.
```

Candidate key constraints:

```
S ( s, sn1, st1, sc1 ) AND S ( s, sn2, st2, sc2 )
                       ⇒   sn1 = sn2 AND
                           st1 = st2 AND
                           sc1 = sc2
etc.
```

Foreign key constraints:

```
SP ( s, p, q ) ⇒ S ( s, sn, st, sc ) AND
                 P ( p, pn, pl, pw, pc ) AND
                 QTY ( q )
```

And so on. *Note:* We assume for the sake of the exposition that variables appearing on the right-hand side of the implication symbol and not on the left (*sn*, *st*, etc., in the example) are existentially quantified. (All others are universally quantified, as explained in Section C.4.) Technically, we need some Skolem functions; *sn*, for example, should really be replaced by (say) SN(*s*), where SN is a Skolem function.

Note, incidentally, that most of the constraints shown above are not pure clauses in

the sense of Section C.5, because the right-hand side is not just a disjunction of simple terms.

Now let us add some more deductive axioms:

```
S ( s, sn, st, sc ) AND st > 15
                    ⇒ GOOD_SUPPLIERS ( s, st, sc )
```

(compare the sample view definition in Chapter 17, Section 17.1).

```
S ( sx, sxn, sxt, sc ) AND S ( sy, syn, syt, sc )
                                ⇒ SS_COLOCATED ( sx, sy )

S ( s, sn, st, c ) AND P ( p, pn, pl, pw, c )
                                ⇒ SP_COLOCATED ( s, p )
```

And so on.

In order to make the example a little more interesting, let us now extend the suppliers-and-parts database to incorporate a part structure relation, showing which parts *px* contain which parts *py* as immediate (i.e. first-level) components. First a constraint to show that *px* and *py* must both identify existing parts:

```
PART_STRUCTURE ( px, py ) ⇒ P ( px, xn, xl, xw, xc ) AND
                             P ( py, yn, yl, yw, yc )
```

Some data values:

```
PART_STRUCTURE ( P1, P2 )
PART_STRUCTURE ( P1, P3 )
PART_STRUCTURE ( P2, P3 )
PART_STRUCTURE ( P2, P4 )
etc.
```

(In practice PART_STRUCTURE would probably also have a "quantity" argument, showing how many *py*'s it takes to make a *px*, but we omit this refinement for simplicity.)

Now we can add a pair of deductive axioms to explain what it means for part *px* to contain part *py* as a component *at any level:*

```
PART_STRUCTURE ( px, py ) ⇒ COMPONENTOF ( px, py )

PART_STRUCTURE ( px, pz ) AND COMPONENTOF ( pz, py )
                          ⇒ COMPONENTOF ( px, py )
```

In other words, part *py* is a component of part *px* (at some level) if it is either an immediate component of part *px* or an immediate component of some part *pz* that is in turn a component (at some level) of part *px*. Note that the second axiom here is recursive—it defines the COMPONENTOF predicate in terms of itself. Classical relational systems, by contrast, do not permit view definitions (or queries or integrity constraints or . . .) to be recursive in such a manner. This ability to support recursion is one of the most immediately obvious distinctions between deductive DBMSs and their classical counterparts—although, as mentioned in Section C.5 and also at earlier points in this

book, there is no fundamental reason why the classical relational algebra should not be extended to support an appropriate set of recursive operators.

We will have more to say regarding this recursive capability in the next section.

Datalog

From the foregoing discussion, it should be clear that one of the most directly visible portions of a deductive DBMS will be a language in which the deductive axioms (usually called **rules**) can be formulated. The best known example of such a language is called (by analogy with Prolog) **Datalog** [C.9]. We present a brief discussion of Datalog in this subsection. *Note:* The emphasis in Datalog is on its descriptive power, not its computational power (as was also the case with the original relational model, incidentally). The objective is to define a language that ultimately will have greater expressive power than conventional relational languages [C.9]. As a consequence, the stress in Datalog—indeed, the stress throughout logic-based database systems in general—is very heavily on query, not update, though it is possible, and desirable, to extend the language to support update also (see later).

In its simplest form, Datalog supports the formulation of rules as simple Horn clauses without functions. In Section C.4, we defined a Horn clause to be a WFF of either of the following two forms:

```
A1 AND A2 AND ... AND An

A1 AND A2 AND ... AND An ⇒ B
```

(where the A's and B are nonnegated predicate instances involving only constants and variables). Following the style of Prolog, however, Datalog actually writes the second of these the other way around:

```
B ⇐ A1 AND A2 AND ... AND An
```

To be consistent with other publications in this area, therefore, we will do the same in what follows. In such a clause, B is the **rule head** (or conclusion) and the A's are the **rule body** (or premises or **goal;** each individual A is a **subgoal**). For brevity, the ANDs are often replaced by commas. A **Datalog program** is a set of such clauses separated in some conventional manner—e.g., by semicolons (in this book, however, we will not use semicolons but instead will simply start each new clause on a new line). No meaning attaches to the order of the clauses within such a program.

Note that *the entire "deductive database"* can be regarded as a Datalog program in the foregoing sense. For example, we could take all of the axioms stated above for suppliers-and-parts (the ground axioms, the integrity constraints, and the deductive axioms), write them all in Datalog style, separate them by semicolons or by writing them on separate lines, and the result would be a Datalog program. As noted earlier, however, the extensional part of the database will typically *not* be created in such a fashion, but rather in some more conventional manner. Thus, the primary function of Datalog is to support the formulation of deductive axioms specifically. As already pointed out,

that function can be regarded as an extension of the view definition mechanism found in conventional relational DBMSs today.

Datalog can also be used as a query language (again, much like Prolog). For example, suppose we have been given the following Datalog definition of GOOD_SUPPLIERS:

```
GOOD_SUPPLIERS ( s, st, sc ) ⇐ S ( s, sn, st, sc ) AND st > 15
```

Here are some typical queries against GOOD_SUPPLIERS.

1. Find all good suppliers:
   ```
   ? ⇐ GOOD_SUPPLIERS ( s, st, sc )
   ```
2. Find good suppliers in Paris:
   ```
   ? ⇐ GOOD_SUPPLIERS ( s, st, Paris )
   ```
3. Is supplier S1 a good supplier?
   ```
   ? ⇐ GOOD_SUPPLIERS ( S1, st, sc )
   ```

And so on. In other words, a query in Datalog consists of a special rule with a head of "?" and a body consisting of a single term that denotes the query result; the head "?" means (by convention) "Display."

It should be pointed out that, despite the fact that Datalog does support recursion, there are quite a few features of conventional relational languages that Datalog in its original form does not support—scalar operations ("+", "–", etc.), aggregate operations (COUNT, SUM, etc.), set difference (because clauses cannot be negated), grouping, etc. It also does not support attribute naming (the significance of a predicate argument depends on its relative position), nor does it provide full domain support (i.e., user-defined data type support in the sense of Chapter 19). As noted earlier, it also does not provide any update operations, nor (as a special case of the latter) does it support the declarative specification of foreign key delete and update rules in the sense of Chapter 5 (DELETE CASCADES, etc.).

In order to address some of the foregoing shortcomings, a variety of extensions to basic Datalog have been proposed. Those extensions are intended to provide the following features, among others:

- *Negative premises*—for example:
  ```
  SS_COLOCATED ( sx, sy ) ⇐ S ( sx, sxn, sxt, sc ) AND
                            S ( sy, syn, syt, sc ) AND
                            NOT ( sx = sy )
  ```
- *Scalar functions* (builtin and user-defined)—for example:
  ```
  P_WT_IN_GRAMS ( p, pn, pl, pg, pc ) ⇐
                  P ( p, pn, pl, pw, pc ) AND pg = pw * 454
  ```
  In this example we have assumed that the builtin function "*" (multiplication) can be written in conventional infix notation. A more orthodox logic representation of the term following the AND would be "=(*pg*,*(*pw*,454))".
- *Aggregate functions and grouping* (somewhat along the lines of our relational SUMMARIZE operator—see Chapter 6). Such operators are necessary in order to

address (for example) what is sometimes called the *gross requirements* problem, which is the problem of finding, not only which parts *py* are components of some part *px* at any level, but also *how many py*'s (at all levels) it takes to make a *px* (naturally we are assuming here that PART_STRUCTURE includes a QTY attribute).

- *Update operations:* One approach to meeting this obvious requirement [C.10]—not the only one—is based on the observation that in basic Datalog, any predicate in a rule head must be nonnegated, and that every tuple generated by the rule can be regarded as being "inserted" into the result relation. A possible extension would thus be to allow negated predicates in a rule head and to treat the negation as requesting the *deletion* (of pertinent tuples).
- *NonHorn clauses in the rule body*—in other words, allow completely general WFFs in the definition of rules.

A survey of the foregoing extensions, with examples, can be found in the book by Gardarin and Valduriez [C.10], which also discusses a variety of Datalog implementation techniques.

C.7 Recursive Query Processing

As indicated in the previous section, one of the most notable features of deductive database systems is their support for recursion (recursive rule definitions, and hence recursive queries also). As a consequence of this fact, the past few years have seen a great deal of research into techniques for implementing such recursion—indeed, just about every database conference since 1986 or so has included one or more papers on the subject (see the References and Bibliography section at the end of this appendix). Since recursive query support represents a problem that typically has not existed in classical DBMSs, we discuss it briefly in the present section.

By way of example, we repeat from Section C.6 the recursive definition of the "part structure" relation (for brevity, however, we now abbreviate PART_STRUCTURE to PS and COMPONENTOF to COMP; we also convert the definition to Datalog form).

```
COMP ( px, py ) ⇐ PS ( px, py )

COMP ( px, py ) ⇐ PS ( px, pz ) AND COMP ( pz, py )
```

Here is a typical recursive query against this database ("Explode part P1"):

```
? ⇐ COMP ( P1, py )
```

To return to the definition *per se:* The second rule in that definition—i.e., the recursive rule—is said to be **linearly** recursive because the predicate in the rule head appears just once in the rule body. As a matter of fact, it would be possible to restate the definition in such a way that the recursion would not be linear:

```
COMP ( px, py ) ⇐ PS ( px, py )

COMP ( px, py ) ⇐ COMP ( px, pz ) AND COMP ( pz, py )
```

However, there is a general feeling that linear recursion represents "the interesting case," in the sense that most recursions that arise in practice are naturally linear, and furthermore there are known efficient techniques for dealing with the linear case [C.16]. We therefore restrict our attention to linear recursion for the remainder of this section.

Note: For completeness, we should point out that it is necessary to generalize the definition of "recursive rule" (and of linear recursion) to deal with more complex cases such as the following:

```
P ( x, y ) ⇐ Q ( x, z ) AND R ( z, y )
Q ( x, y ) ⇐ P ( x, z ) AND S ( z, y )
```

For brevity, we ignore such refinements here. The interested reader is referred to reference [C.16] for more details.

As in classical (nonrecursive) query processing, the overall problem of implementing a given recursive query can be divided into two subproblems, namely (a) transforming the original query into some equivalent but more efficient form and (b) actually executing the result of that transformation. The literature contains descriptions of a variety of attacks on both of these problems. Reference [C.16] provides an excellent survey, analysis, and comparison of published techniques as of about 1988. In the present section, we briefly discuss some of the simpler techniques. We will illustrate them by showing their application to the query "Explode part P1," using the following set of sample values for the PS relation (based on Fig. 4.4 from Chapter 4, but ignoring QTY for simplicity):

PS

PX	PY
P1	P2
P1	P3
P2	P3
P2	P4
P3	P5
P4	P5
P5	P6

Unification and Resolution

One possible approach, of course, is to use the standard Prolog techniques of **unification and resolution,** as described in Section C.4. In the example, this approach works as follows. The first premises are the deductive axioms, which look like this in conjunctive normal form:

1. `NOT PS ( px, py ) OR COMP ( px, py )`
2. `NOT PS ( px, pz ) OR NOT COMP ( pz, py ) OR COMP ( px, py )`

We construct another premise from the desired conclusion:

3. NOT COMP (P1, *py*) OR RESULT (*py*)

The ground axioms form the remaining premises. Consider, for example, the ground axiom

4. PS (P1, P2)

Substituting P1 for *px* and P2 for *py* in line 1, we can resolve lines 1 and 4 to yield

5. COMP (P1, P2)

Now substituting P2 for *py* in line 3 and resolving lines 3 and 5, we obtain

6. RESULT (P2)

So P2 is a component of P1. An exactly analogous argument will show that P3 is also a component of P1. Now, of course, we have the additional axioms COMP(P1,P2) and COMP(P1,P3); we can now apply the foregoing process recursively to determine the complete part explosion. The details are left as an exercise for the reader.

In practice, however, unification and resolution can be quite costly in performance. It will thus often be desirable to find some more efficient strategy. The remaining subsections below discuss some possible approaches to this problem.

Naive Evaluation

Naive evaluation [C.25] is probably the simplest approach of all. As the name suggests, the algorithm is very simple-minded; it can most easily be explained (for our sample query) in terms of the following pseudocode.

```
COMP := PS ;
do until COMP reaches a "fixpoint" ;
   COMP := COMP UNION ( COMP ⋈ PS ) ;
end ;
DISPLAY := COMP WHERE PX = 'P1' ;
```

Relations COMP and DISPLAY (like relation PS) each have two attributes, PX and PY. Loosely speaking, the algorithm works by repeatedly forming an intermediate result consisting of the union of the join of relation PS and the previous intermediate result, until that intermediate result reaches a **"fixpoint"**—i.e., until it ceases to grow. *Note:* The expression "COMP ⋈ PS" is shorthand for "join COMP and PS over COMP.PY and PS.PX and project the result over COMP.PX and PS.PY." For brevity, we ignore the attribute renaming operations that our dialect of the algebra would require to make this work (see Chapter 6).

Let us step through the algorithm with our sample set of data values. After the first iteration of the loop, the value of the expression COMP ⋈ PS is as shown below on the left and the resulting value of COMP is as shown below on the right (with tuples added on this iteration flagged with an asterisk):

COMP ⋈ PS

PX	PY
P1	P3
P1	P4
P1	P5
P2	P5
P3	P6
P4	P6

COMP

PX	PY	
P1	P2	
P1	P3	
P2	P3	
P2	P4	
P3	P5	
P4	P5	
P5	P6	
P1	P4	*
P1	P5	*
P2	P5	*
P3	P6	*
P4	P6	*

After the second iteration, they look like this:

COMP ⋈ PS

PX	PY
P1	P3
P1	P4
P1	P5
P2	P5
P3	P6
P4	P6
P1	P6
P2	P6

COMP

PX	PY	
P1	P2	
P1	P3	
P2	P3	
P2	P4	
P3	P5	
P4	P5	
P5	P6	
P1	P4	
P1	P5	
P2	P5	
P3	P6	
P4	P6	
P1	P6	*
P2	P6	*

Note carefully that the computation of COMP ⋈ PS in this second step repeats the entire computation of COMP ⋈ PS from the first step but additionally computes some extra tuples (actually just two extra tuples—(P1,P6) and (P2,P6)—in the case at hand). This is one reason why the naive evaluation algorithm is not very intelligent.

After the third iteration, the value of COMP ⋈ PS (after more repeated computation) turns out to be the same as on the previous iteration; COMP has thus reached a fixpoint, and we exit from the loop. The final result is then computed as a restriction of COMP:

COMP

PX	PY
P1	P2
P1	P3
P1	P4
P1	P5
P1	P6

Another glaring inefficiency is now apparent: The algorithm has effectively computed the explosion for *every* part—in fact, it has computed the **transitive closure** of relation PS (see below)—and has then thrown everything away again except for the tuples actually wanted. In other words, again, a great deal of unnecessary work has been performed.

A note regarding transitive closure: The **transitive closure** of a binary relation $R\{X,Y\}$ is a superset of R, defined as follows: The tuple (x,y) appears in the transitive closure of R if and only if it appears in R or there exists a sequence of values $z1$, $z2$, . . . zn such that the tuples $(x,z1)$, $(z1,z2)$, . . . (zn,y) all appear in R.

Note in conclusion that the naive evaluation technique can be regarded as an application of forward chaining: Starting from the extensional database (i.e., the actual data values), it applies the premises of the definition (i.e., the rule body) repeatedly until the desired result is obtained. In fact, the algorithm actually computes the *minimal model* for the Datalog program (see Sections C.5 and C.6).

Seminaive Evaluation

The first obvious improvement to the naive evaluation algorithm is to avoid repeating the computations of each step in the next step: **semi**naive evaluation [C.28]. In other words, in each step we compute just the new tuples that need to be appended on this particular iteration. Again we explain the idea in terms of the "Explode part P1" example. Pseudocode:

```
NEW  := PS ;
COMP := NEW ;
do until NEW is empty ;
   NEW  := ( NEW ※ PS ) MINUS COMP ;
   COMP := COMP UNION NEW ;
end ;
DISPLAY := COMP WHERE PX = 'P1' ;
```

Let us again step through the algorithm. On initial entry into the loop, NEW and COMP are both identical to PS:

NEW

PX	PY
P1	P2
P1	P3
P2	P3
P2	P4
P3	P5
P4	P5
P5	P6

COMP

PX	PY
P1	P2
P1	P3
P2	P3
P2	P4
P3	P5
P4	P5
P5	P6

At the end of the first iteration, they look like this:

NEW

PX	PY
P1	P4
P1	P5
P2	P5
P3	P6
P4	P6

COMP

PX	PY	
P1	P2	
P1	P3	
P2	P3	
P2	P4	
P3	P5	
P4	P5	
P5	P6	
P1	P4	*
P1	P5	*
P2	P5	*
P3	P6	*
P4	P6	*

COMP is the same as it was at this stage under naive evaluation, and NEW is just the new tuples that were added to COMP on this iteration; note in particular that NEW does *not* include the tuple (P1,P3) (compare the naive evaluation counterpart).

At the end of the next iteration we have:

NEW

PX	PY
P1	P6
P2	P6

COMP

PX	PY	
P1	P2	
P1	P3	
P2	P3	
P2	P4	
P3	P5	
P4	P5	
P5	P6	
P1	P4	
P1	P5	
P2	P5	
P3	P6	
P4	P6	
P1	P6	*
P2	P6	*

The next iteration makes NEW empty, and so we leave the loop.

Static Filtering

Static filtering is a refinement on the basic idea from classical optimization theory of performing restrictions as early as possible. It can be regarded as an application of backward chaining, in that it effectively uses information from the query (the conclusion) to modify the rules (the premises). It is also referred to as *reducing the set of relevant facts*, in that it (again) uses information from the query to eliminate useless tuples in the extensional database right at the outset. The effect in terms of our example can be explained in terms of the following pseudocode:

```
NEW  := PS WHERE PX = 'P1' ;
COMP :=  NEW ;
do until NEW is empty ;
   NEW  := ( NEW ⨯ PS ) MINUS COMP ;
   COMP := COMP UNION NEW ;
end ;
DISPLAY := COMP ;
```

Once again we step through the algorithm. On initial entry into the loop, NEW and COMP both look like this:

NEW

PX	PY
P1	P2
P1	P3

COMP

PX	PY
P1	P2
P1	P3

At the end of the first iteration, they look like this:

NEW

PX	PY
P1	P4
P1	P5

COMP

PX	PY	
P1	P2	
P1	P3	
P1	P4	*
P1	P5	*

At the end of the next iteration we have:

NEW

PX	PY
P1	P6

COMP

PX	PY	
P1	P2	
P1	P3	
P1	P4	
P1	P5	
P1	P6	*

The next iteration makes NEW empty, and so we leave the loop.

This concludes our brief introduction to recursive query processing strategies. Of course, many other approaches have been proposed in the literature, most of them considerably more sophisticated than the rather simple ones discussed above; however, there is insufficient space in a book of this nature to cover all of the background material that is needed for a proper understanding of those approaches. See references [C.16–C.43] for further discussion.

C.8 Summary

This brings us to the end of our short introduction to the topic of database systems that are based on logic. The idea is still comparatively new, but it looks very interesting. Several potential advantages were identified at various points in the preceding sections.

One further advantage, not mentioned explicitly in the body of the appendix, is that logic could form the basis of a genuinely seamless integration between general-purpose programming languages and the database. In other words, instead of the "embedded data sublanguage" approach supported by most DBMS products today—an approach that is not particularly elegant, to say the least—the system could provide a single logic-based language in which "data is data," regardless of whether it is kept in a shared database or local to the application. (Of course, there are a number of obstacles to be overcome before such a goal can be achieved, not the least of which is to demonstrate to the satisfaction of the IT community at large that logic is suitable as a basis for a general-purpose programming language in the first place.)

Quite apart from the foregoing possibility, it is certain that some applications, at least, will need Datalog-style access (or similar) to data stored in a shared database. The question thus arises: How exactly can such access be provided? There are two broad answers to this question, referred to generically as *loose coupling* and *tight coupling* [C.11].

- The loose coupling approach consists essentially of taking an existing DBMS and an existing logic programming language system and providing a call interface between the two. In such an approach, the user is definitely aware of the fact that there are two distinct systems involved—database operations (e.g., SQL statements) must be executed to retrieve data before logic operations (e.g., Prolog rules) can be applied to that data. This approach thus certainly does *not* provide the "seamless integration" referred to above.
- The tight coupling approach, by contrast, involves a proper integration of the DBMS and the logic programming system; in other words, the DBMS query language includes direct support for the logical inferencing operations (etc.). Thus, the user deals with one language, not two.

The pros and cons of the two approaches are obvious: Loose coupling is more straightforward to implement but is not so attractive from the user's point of view; conversely, tight coupling is harder to implement but is more attractive to the user. In addition, tight coupling is likely to perform better (the scope for logic query optimization is necessarily very limited in a loose coupling system). For obvious reasons, however, the first commercial products (a few do exist) are based on loose coupling.

Let us quickly review the major points of the material we have covered. We began with a brief tutorial on **propositional calculus** and **predicate calculus,** introducing the following concepts among others:

- An **interpretation** of a set of WFFs is the combination of (a) a universe of discourse, (b) a mapping from individual constants appearing in those WFFs to objects in that universe, and (c) a set of defined meanings for the predicates and functions appearing in those WFFs.
- A **model** for a set of WFFs is an interpretation for which all WFFs in the set evaluate to *true*. A given set of WFFs can have any number of models (in general).
- A **proof** is the process of showing that some given WFF *g* (the **conclusion**) is a

logical consequence of some given set of WFFs *f1, f2, . . . fn* (the **premises**). We discussed one proof method, known as **resolution and unification,** in some detail.

We then examined the **proof-theoretic** view of databases. In such a view, the database is regarded as consisting of the combination of an **extensional** database and an **intensional** database. The extensional database contains **ground axioms,** i.e., the base data (loosely speaking); the intensional database contains integrity constraints and **deductive axioms,** i.e., views (again, loosely speaking). The "meaning" of the database then consists of the set of **theorems** that can be deduced from the axioms; executing a query becomes (at least conceptually) a **theorem-proving** process. A **deductive DBMS** is a DBMS that supports this proof-theoretic view. We briefly described **Datalog,** a user language for such a DBMS.

One immediately obvious distinction between Datalog and traditional relational languages is that Datalog supports **recursive** axioms, and hence recursive queries—though there is no reason why the traditional relational algebra and calculus should not be extended to do likewise.* We discussed some simple techniques for evaluating such queries.

In conclusion: We opened this appendix by mentioning a number of terms—"logic database," "inferential DBMS," "deductive DBMS," etc., etc.—that are often met with nowadays in the research literature (and indeed in vendor advertising). Let us therefore close it by providing some definitions for those terms. We warn the reader, however, that there is not always a consensus on these matters, and different definitions can probably be found in other publications. Following are the definitions preferred by the present writer.

- *Recursive query processing:* This is an easy one. Recursive query processing refers to the evaluation (and in particular optimization) of queries whose definition is intrinsically recursive. See Section C.7.
- *Knowledge base:* This term is sometimes used to mean what we called the intensional database in Section C.6—i.e., it consists of the *rules* (the integrity constraints and deductive axioms), as opposed to the base data, which constitutes the extensional database. But then other writers use "knowledge base" to mean the combination of both the intensional and extensional databases (see "deductive database" below)—except that, as reference [C.10] puts it, "a knowledge base often includes complex objects [as well as] classical relations." (See Part VI of this book for a discussion of "complex objects.") Then again, the term has another, more specific meaning altogether in natural language systems. It is probably best to avoid the term entirely.
- *Knowledge:* Another easy one! Knowledge is what is in the knowledge

* In this connection it is interesting to observe that relational DBMSs need to be able to perform recursive processing under the covers anyway, because the catalog will contain certain recursively structured information (e.g., domain definitions expressed in terms of other domain definitions, view definitions expressed in terms of other view definitions, etc.).

base . . . This definition thus reduces the problem of defining "knowledge" to a previously unsolved problem.

- *Knowledge base management system (KBMS):* The software that manages the knowledge base. The term is typically used as a synonym for deductive DBMS, *q.v.*
- *Deductive DBMS:* A DBMS that supports the proof-theoretic view of databases, and in particular is capable of deducing additional information from the extensional database by applying inferential (i.e., deductive) rules that are stored in the intensional database. A deductive DBMS will almost certainly support recursive rules and so perform recursive query processing.
- *Deductive database:* (Deprecated term.) A database that is managed by a deductive DBMS.
- *Expert DBMS:* Synonym for deductive DBMS.
- *Expert database:* (Deprecated term.) A database that is managed by an expert DBMS.
- *Inferential DBMS:* Synonym for deductive DBMS.
- *Logic-based system:* Synonym for deductive DBMS.
- *Logic database:* (Deprecated term.) Synonym for deductive database.
- *Logic as a data model:* A data model consists of objects, integrity rules, and operators. In a deductive DBMS, the objects, integrity rules, and operators are all represented in the same uniform way, namely as axioms in a logic language such as Datalog; indeed, as explained in Section C.6, a database in such a system can be regarded, precisely, as a logic program containing axioms of all three kinds. In such a system, therefore, we might legitimately say that the abstract data model for the system *is* logic itself.

Exercises

C.1 Use the resolution method to see whether the following statements are valid in the propositional calculus:

(a) `A ⇒ B, C ⇒ B, D ⇒ ( A OR C ), D ⊢ B`

(b)
```
( A ⇒ B ) AND ( C ⇒ D ), ( B ⇒ E AND D ⇒ F ),
    NOT ( E AND F ), A ⇒ C ⊢ NOT A
```

(c)
```
( A OR B ) ⇒ D, D ⇒ NOT ( E OR F ),
    NOT ( B AND C AND E ) ⊢
                NOT ( G ⇒ NOT ( C AND H ) )
```

C.2 Convert the following WFFs to clausal form:

```
(a) FORALL x ( FORALL y ( p ( x, y ) ⇒ EXISTS z ( q ( x, z ) ) ) )
(b) EXISTS x ( EXISTS y ( p ( x, y ) ⇒ FORALL z ( q ( x, z ) ) ) )
(c) EXISTS x ( EXISTS y ( p ( x, y ) ⇒ EXISTS z ( q ( x, z ) ) ) )
```

C.3 The following is a fairly standard example of a logic database:

```
MAN    ( Adam )
WOMAN  ( Eve )
MAN    ( Cain )
MAN    ( Abel )
MAN    ( Enoch )

PARENT ( Adam, Cain )
PARENT ( Adam, Abel )
PARENT ( Eve, Cain )
PARENT ( Eve, Abel )
PARENT ( Cain, Enoch )

FATHER ( x, y ) ⇐ PARENT ( x, y ) AND MAN ( x )
MOTHER ( x, y ) ⇐ PARENT ( x, y ) AND WOMAN ( x )

SIBLING ( x, y ) ⇐ PARENT ( z, x ) AND PARENT ( z, y )

BROTHER ( x, y ) ⇐ SIBLING ( x, y ) AND MAN ( x )

SISTER ( x, y ) ⇐ SIBLING ( x, y ) AND WOMAN ( x )

ANCESTOR ( x, y ) ⇐ PARENT ( x, y )
ANCESTOR ( x, y ) ⇐ PARENT ( x, z ) AND ANCESTOR ( z, y )
```

Use the resolution method to answer the following queries:

(a) Who is the mother of Cain?

(b) Who are Cain's siblings?

(c) Who are Cain's brothers?

(d) Who are Cain's sisters?

(e) Who are Enoch's ancestors?

C.4 Define the terms *interpretation* and *model*.

C.5 Write a set of Datalog axioms for the definitional portion (only) of the suppliers-parts-projects database.

C.6 Give Datalog solutions, where possible, to Exercises 6.13–6.48.

C.7 Give Datalog solutions, where possible, to Exercises 16.1–16.16.

C.8 Complete (to your own satisfaction) the explanation given in Section C.7 of the unification and resolution implementation of the "Explode part P1" query.

References and Bibliography

The field of logic-based systems has mushroomed within the last few years. The following list represents a tiny fraction of the literature currently available. It is partially arranged into groups, as follows. First, references [C.1–C.9] are books that either are devoted to the subject of logic in general (particularly in a computing and/or database context) or are collections of papers on logic-based database systems specifically. References [C.10–C.12] are tutorials, as are the books by Ceri *et al.* [C.46] and Das [C.47]. References [C.14], [C.17–C.20], [C.30], and [C.49–C.50] are concerned with the transitive closure operation and its implementation. References

[C.21–C.24] describe an important recursive query processing technique called "magic sets" (and variations thereon). The remaining references are included principally to show just how much investigation is going on in this field; they address a variety of aspects of the subject, and are presented for the most part without further comment.

C.1 Robert R. Stoll. *Sets, Logic, and Axiomatic Theories*. San Francisco, Calif.: W. H. Freeman and Company (1961).

A good introduction to logic in general.

C.2 Zohar Manna and Richard Waldinger. *The Logical Basis for Computer Programming. Volume I: Deductive Reasoning* (1985); *Volume II: Deductive Techniques* (1990). Reading, Mass.: Addison-Wesley (1985, 1990).

C.3 Peter M. D. Gray. *Logic, Algebra and Databases*. Chichester, England: Ellis Horwood Ltd. (1984).

Contains a good gentle introduction to propositional calculus and predicate calculus (among a number of other relevant topics) from a database point of view.

C.4 Adrian Walker, Michael McCord, John F. Sowa, and Walter G. Wilson. *Knowledge Systems and Prolog* (2nd edition). Reading, Mass.: Addison-Wesley (1990).

This book is about logic programming in general, not logic-based database systems specifically, but it contains a great deal that is relevant to the latter topic.

C.5 Hervé Gallaire and Jack Minker. *Logic and Data Bases*. New York, N.Y.: Plenum Publishing Corp. (1978).

One of the first, if not *the* first, collections of papers in the field.

C.6 Larry Kershberg (ed.). *Expert Database Systems* (Proc. 1st International Workshop on Expert Database Systems, Kiawah Island, S.C.). Menlo Park, Calif.: Benjamin/Cummings (1986).

An excellent and thought-provoking collection of papers. Not all of them are directly related to the main subject of the present appendix, however. Indeed, the titles of the sections betray a certain degree of confusion as to what the subject of "expert database systems" really is! Those titles are as follows:

1. Theory of knowledge bases
2. Logic programming and databases
3. Expert database system architectures, tools, and techniques
4. Reasoning in expert database systems
5. Intelligent database access and interaction

In addition, there is a keynote paper by John Smith on expert database systems, and reports from working groups on (1) knowledge base management systems, (2) logic programming and databases, and (3) object-oriented database systems and knowledge systems. As Kershberg remarks in his preface, the expert database system concept "connotes diverse definitions and decidedly different architectures."

C.7 Jack Minker (ed.). *Foundations of Deductive Databases and Logic Programming*. San Mateo, Calif.: Morgan Kaufmann (1988).

C.8 John Mylopoulos and Michael L. Brodie (eds.). *Readings in Artificial Intelligence and Databases*. San Mateo, Calif.: Morgan Kaufmann (1988).

C.9 Jeffrey D. Ullman. *Database and Knowledge-Base Systems* (two volumes). Rockville, Md.: Computer Science Press (1988, 1989).

Volume I of this two-volume work includes one (long) chapter (out of a total of 10 chap-

ters) that is entirely devoted to the logic-based approach. That chapter (which is the origin of Datalog, incidentally) includes a discussion of the relationship between logic and relational algebra, and another on relational calculus—both domain and tuple versions—as a special case of the logic approach. Volume II includes five chapters (out of seven) on various aspects of logic-based systems.

C.10 Georges Gardarin and Patrick Valduriez. *Relational Databases and Knowledge Bases.* Reading, Mass.: Addison-Wesley (1989).

Contains a chapter on deductive systems that (although tutorial in nature) goes into the underlying theory, optimization algorithms, etc., in much more detail than the present appendix does.

C.11 Michael Stonebraker. *Introduction to* "Integration of Knowledge and Data Management." In M. Stonebraker (ed.): *Readings in Database Systems.* San Mateo, Calif.: Morgan Kaufmann (1988).

C.12 Hervé Gallaire, Jack Minker, and Jean-Marie Nicolas. "Logic and Databases: A Deductive Approach." *ACM Comp. Surv. 16,* No. 2 (June 1984).

C.13 Veronica Dahl. "On Database Systems Development through Logic." *ACM TODS 7,* No. 1 (March 1982).

A good and clear description of the basic ideas underlying logic-based database systems, with examples taken from a Prolog-based prototype implemented by Dahl in 1977.

C.14 Rakesh Agrawal: "Alpha: An Extension of Relational Algebra to Express a Class of Recursive Queries." *IEEE Transactions on Software Engineering 14,* No. 7 (July 1988).

Proposes a new operator called *alpha* that supports the formulation of "a large class of recursive queries" (actually a superset of linear recursive queries) while staying within the framework of conventional relational algebra (more or less; the operator actually produces a non1NF result, i.e., a relation one of whose attributes is relation-valued, but then places bounds on how that result can be used in order to avoid having to define a complete algebra for such unnormalized relations). The contention is that the *alpha* operator is sufficiently powerful to deal with most practical problems involving recursion, while at the same time being easier to implement efficiently than any completely general recursion mechanism would be. The paper gives several examples of the use of the proposed operator; in particular, it shows how the transitive closure and "gross requirements" problems (see reference [C.17] and Section C.6 respectively) can both easily be handled.

Reference [C.19] describes some related work on implementation. Reference [C.18] is also relevant.

C.15 Raymond Reiter. "Towards a Logical Reconstruction of Relational Database Theory." In *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases, and Programming Languages* (eds., Michael L. Brodie, John Mylopoulos, and Joachim W. Schmidt). New York, N.Y.: Springer-Verlag (1984).

As mentioned in Section C.2, Reiter's work was by no means the first in this area—many researchers had investigated the relationship between logic and databases before (see, e.g., references [C.5], [C.7], and [C.13]). However, it seems to have been Reiter's "logical reconstruction of relational theory" that spurred much of the current activity and high degree of interest in the field.

C.16 François Bancilhon and Raghu Ramakrishnan. "An Amateur's Introduction to Recursive Query Processing Strategies." Proc. 1986 ACM SIGMOD International Conference on Management of Data, Washington, D.C. (May 1986). Republished in revised form in M. Stonebraker (ed.), *Readings in Database Systems.* San Mateo, Calif.: Morgan Kaufmann (1988). Also republished in reference [C.8].

An excellent overview. The paper starts by observing that there is both a positive and a negative side to all of the research on the recursive query implementation problem. The positive side is that numerous techniques have been identified that do at least solve the problem; the negative side is that it is not at all clear how to choose the technique that is most appropriate in a given situation (in particular, most of the techniques are presented in the literature with little or no discussion of performance characteristics). Then, after a section describing the basic ideas of logic databases, the paper goes on to describe a number of proposed algorithms—naive evaluation, seminaive evaluation, iterative query/subquery, recursive query/subquery, APEX, Prolog, Henschen/Naqvi, Aho-Ullman, Kifer-Lozinskii, counting, magic sets, and generalized magic sets. The paper compares these different approaches on the basis of application domain (i.e., the class of problems to which the algorithm can be applied), performance, and ease of implementation. The paper also includes performance figures (with comparative analysis) from testing the various algorithms on a simple benchmark.

C.17 Yannis E. Ioannidis. "On the Computation of the Transitive Closure of Relational Operators." Proc. 12th International Conference on Very Large Data Bases, Kyoto, Japan (August 1986).

Transitive closure is an operation of fundamental importance in recursive query processing [C.18]. This paper proposes a new algorithm (based on a "divide and conquer" approach) for implementing that operation. See also references [C.14], [C.18–C.20], and [C.49–C.50].

C.18 H. V. Jagadish, Rakesh Agrawal, and Linda Ness. "A Study of Transitive Closure as a Recursion Mechanism." Proc. 1987 ACM SIGMOD International Conference on Management of Data, San Francisco, Calif. (May 1987).

To quote from the abstract: "[This paper shows] that every linearly recursive query can be expressed as a transitive closure possibly preceded and followed by operations already available in relational algebra." The suggestion is that providing an efficient implementation of transitive closure is therefore sufficient as a basis for providing an efficient implementation of linear recursion in general, and hence for making deductive DBMSs efficient on a large class of recursive problems.

C.19 Rakesh Agrawal and H. Jagadish. "Direct Algorithms for Computing the Transitive Closure of Database Relations." Proc. 13th International Conference on Very Large Data Bases, Brighton, England (September 1987).

Proposes a set of transitive closure algorithms that "do not view the problem as one of evaluating a recursion, but rather obtain the closure from first principles" (hence the term *direct*). The paper includes a useful summary of earlier work on other direct algorithms.

C.20 Hongjun Lu. "New Strategies for Computing the Transitive Closure of a Database Relation." Proc. 13th International Conference on Very Large Data Bases, Brighton, England (September 1987).

More algorithms for transitive closure. Like reference [C.19], the paper also includes a useful survey of earlier approaches to the problem.

C.21 François Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. "Magic Sets and Other Strange Ways to Implement Logic Programs." Proc. 5th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems (1986).

The basic idea of "magic sets" is to introduce new sets of rules ("magic rules") dynamically that are guaranteed to produce the same result as the original query but are more efficient, in the sense that they reduce the set of "relevant facts" (see Section C.7). The details are a little complex, and beyond the scope of these notes; the reader is referred to

the paper or to Bancilhon and Ramakrishnan's survey [C.16] or the books by Ullman [C.9] or Gardarin and Valduriez [C.10] for more explanation. We remark, however, that numerous variations on the basic idea have been devised—see, e.g., references [C.22–C.24] below.

See also reference [18.22] in Chapter 18.

C.22 Catriel Beeri and Raghu Ramakrishnan. "On the Power of Magic." Proc. 6th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems (1987).

C.23 Domenico Saccà and Carlo Zaniolo. "Magic Counting Methods." Proc. 1987 ACM SIGMOD International Conference on Management of Data, San Francisco, Calif. (May 1987).

C.24 Georges Gardarin. "Magic Functions: A Technique to Optimize Extended Datalog Recursive Programs." Proc. 13th International Conference on Very Large Data Bases, Brighton, England (September 1987).

C.25 A. Aho and J.D. Ullman. "Universality of Data Retrieval Languages." Proc. 6th ACM Symposium on Principles of Programming Languages, San Antonio, Texas (January 1979).

Given a sequence of relations R, $f(R)$, $f(f(R))$, . . . (where f is some fixed function), the **least fixpoint** of the sequence is defined to be a relation R^* derived in accordance with the following naive evaluation algorithm (see Section C.7):

```
R* := R ;
do until R* stops growing ;
   R* := R* UNION f(R*) ;
end ;
```

This paper proposes the addition of a least fixpoint operator to the relational algebra.

C.26 Jeffrey D. Ullman. "Implementation of Logical Query Languages for Databases." *ACM TODS 10*, No. 3 (September 1985).

Describes an important class of implementation techniques for possibly recursive queries. The techniques are defined in terms of "capture rules" on "rule/goal trees," which are graphs that represent a query strategy in terms of clauses and predicates. The paper defines several such rules—one that corresponds to the application of relational algebra operators, two more that correspond to forward and backward chaining respectively, and a "sideways" rule that allows results to be passed from one subgoal to another. Sideways information passing later became the basis for the so-called *magic set* techniques [C.21–C.24].

C.27 Shalom Tsur and Carlo Zaniolo. "LDL: A Logic-Based Data-Language." Proc. 12th International Conference on Very Large Data Bases, Kyoto, Japan (August 1986).

LDL includes (1) sets as a primitive data object, (2) negation (based on set difference), (3) data definition operations, and (4) update operations (for data definitions, base relations, and derived relations—the last of these not implemented at the time of publication of the paper). It is a pure logic language (no ordering dependencies among statements) and is compiled, not interpreted.

See also the book by Naqvi and Tsur [C.45] on the same subject.

C.28 François Bancilhon. "Naive Evaluation of Recursively Defined Relations." In M. Brodie and J. Mylopoulos (eds.): *On Knowledge Base Management Systems: Integrating Database and AI Systems*. New York, N.Y.: Springer-Verlag (1986).

C.29 Eliezer L. Lozinskii. "A Problem-Oriented Inferential Database System." *ACM TODS 11*, No. 3 (September 1986).

The source of the concept of "relevant facts." The paper describes a prototype system that

makes use of the extensional database to curb the otherwise very fast expansion of the search space that inferential techniques typically give rise to.

C.30 Arnon Rosenthal *et al.*. "Traversal Recursion: A Practical Approach to Supporting Recursive Applications." Proc. 1986 ACM SIGMOD International Conference on Management of Data, Washington, D.C. (June 1986).

C.31 Georges Gardarin and Christophe de Maindreville. "Evaluation of Database Recursive Logic Programs as Recurrent Function Series." Proc. 1986 ACM SIGMOD International Conference on Management of Data, Washington, D.C. (June 1986).

C.32 Louiqa Raschid and Stanley Y. W. Su. "A Parallel Processing Strategy for Evaluating Recursive Queries." Proc. 12th International Conference on Very Large Data Bases, Kyoto, Japan (August 1986).

C.33 Nicolas Spyratos. "The Partition Model: A Deductive Database Model." *ACM TODS 12*, No. 1 (March 1987).

C.34 Jiawei Han and Lawrence J. Henschen. "Handling Redundancy in the Processing of Recursive Queries." Proc. 1987 ACM SIGMOD International Conference on Management of Data, San Francisco, Calif. (May 1987).

C.35 Weining Zhang and C. T. Yu. "A Necessary Condition for a Doubly Recursive Rule to be Equivalent to a Linear Recursive Rule." Proc. 1987 ACM SIGMOD International Conference on Management of Data, San Francisco, Calif. (May 1987).

C.36 Wolfgang Nejdl. "Recursive Strategies for Answering Recursive Queries—The RQA/FQI Strategy." Proc. 13th International Conference on Very Large Data Bases, Brighton, England (September 1987).

C.37 Kyu-Young Whang and Shamkant B. Navathe. "An Extended Disjunctive Normal Form Approach for Optimizing Recursive Logic Queries in Loosely Coupled Environments." Proc. 13th International Conference on Very Large Data Bases, Brighton, England (September 1987).

C.38 Jeffrey F. Naughton. "Compiling Separable Recursions." Proc. 1988 ACM SIGMOD International Conference on Management of Data, Chicago, Ill. (June 1988).

C.39 Cheong Youn, Lawrence J. Henschen, and Jiawei Han. "Classification of Recursive Formulas in Deductive Databases." Proc. 1988 ACM SIGMOD International Conference on Management of Data, Chicago, Ill. (June 1988).

C.40 S. Ceri, G. Gottlob, and L. Lavazza. "Translation and Optimization of Logic Queries: The Algebraic Approach." Proc. 12th International Conference on Very Large Data Bases, Kyoto, Japan (August 1986).

C.41 S. Ceri and L. Tanca. "Optimization of Systems of Algebraic Equations for Evaluating Datalog Queries." Proc. 13th International Conference on Very Large Data Bases, Brighton, England (September 1987).

C.42 Allen Van Gelder. "A Message Passing Framework for Logical Query Evaluation." Proc. 1986 ACM SIGMOD International Conference on Management of Data, Washington, D.C. (June 1986).

C.43 Ouri Wolfson and Avi Silberschatz. "Distributed Processing of Logic Programs." Proc. 1988 ACM SIGMOD International Conference on Management of Data, Chicago, Ill. (June 1988).

C.44 Jeffrey F. Naughton *et al.* "Efficient Evaluation of Right-, Left-, and Multi-Linear Rules." Proc. 1989 ACM SIGMOD International Conference on Management of Data, Portland, Ore. (June 1989).

C.45 Shamim Naqvi and Shalom Tsur. *A Logical Language for Data and Knowledge Bases.* New York, N.Y.: Computer Science Press (1989).

An indepth, book-length presentation of the language LDL [C.27].

C.46 S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases.* New York, N.Y.: Springer-Verlag (1990).

C.47 Subrata Kumar Das. *Deductive Databases and Logic Programming.* Reading, Mass.: Addison-Wesley (1992).

C.48 Michael Kifer and Eliezer Lozinskii. "On Compile-Time Query Optimization in Deductive Databases by Means of Static Filtering." *ACM TODS 15,* No. 3 (September 1990).

C.49 Rakesh Agrawal, Shaul Dar, and H. V. Jagadish. "Direct Transitive Closure Algorithms: Design and Performance Evaluation." *ACM TODS 15,* No. 3 (September 1990).

C.50 H. V. Jagadish. "A Compression Method to Materialize Transitive Closure." *ACM TODS 15,* No. 4 (December 1990).

Proposes an indexing technique that allows the transitive closure of a given relation to be stored in compressed form, such that testing to see whether a given tuple appears in the closure can be done via a single table lookup followed by an index comparison.

C.51 Serge Abiteboul and Stéphane Grumbach. "A Rule-Based Language with Functions and Sets." *ACM TODS 16,* No. 1 (March 1991).

Describes a language called COL ("complex object language")—an extension of Datalog—that integrates the ideas of deductive and object-oriented databases.

Answers to Selected Exercises

C.1 (a) and (b) are valid, (c) is not.

C.2 In the following, *a*, *b*, and *c* are Skolem constants and *f* is a Skolem function with two arguments.

(a) `p ( x, y ) ⇒ q ( x, f ( x, y ) )`

(b) `p ( a, b ) ⇒ q ( a, z )`

(c) `p ( a, b ) ⇒ q ( a, c )`

C.6 In accordance with our usual practice, we have numbered the following solutions as C.6.*n*, where 6.*n* is the number of the original exercise in Chapter 6.

C.6.13
```
? ⇐ J ( j, jn, jc )
```

C.6.14
```
? ⇐ J ( j, jn, London )
```

C.6.15
```
RES ( s ) ⇐ SPJ ( s, p, J1 )
? ⇐ RES ( s )
```

C.6.16
```
? ⇐ SPJ ( s, p, j, q ) AND 300 ≤ q AND q ≤ 750
```

C.6.17
```
RES ( pl, pc ) ⇐ P ( p, pn, pl, w, pc )
? ⇐ RES ( pl, pc )
```

C.6.18
```
RES ( s, p, j ) ⇐ S ( s, sn, st, c ) AND
                  P ( p, pn, pl, w, c ) AND
                  J ( j, jn, c )
? ⇐ RES ( s, p, j )
```

C.6.19–C.6.20 Cannot be done without negation.

C.6.21
```
RES ( p ) ⇐ SPJ ( s, p, j, q ) AND
            S ( s, sn, st, London )
? ⇐ RES ( p )
```

C.6.22
```
RES ( p ) ⇐ SPJ ( s, p, j, q ) AND
            S ( s, sn, st, London ) AND
            J ( j, jn, London )
? ⇐ RES ( p )
```

C.6.23
```
RES ( c1, c2 ) ⇐ SPJ ( s, p, j, q ) AND
                 S ( s, sn, st, c1 ) AND
                 J ( j, jn, c2 )
? ⇐ RES ( c1, c2 )
```

C.6.24
```
RES ( p ) ⇐ SPJ ( s, p, j, q ) AND
            S ( s, sn, st, c ) AND
            J ( j, jn, c )
? ⇐ RES ( p )
```

C.6.25 Cannot be done without negation.

C.6.26
```
RES ( p1, p2 ) ⇐ SPJ ( s, p1, j1, q1 ) AND
                 SPJ ( s, p2, j2, q2 )
? ⇐ RES ( p1, p2 )
```

C.6.27–C.6.30 Cannot be done without grouping and aggregate functions.

C.6.31
```
RES ( jn ) ⇐ J ( j, jn, jc ) AND
             SPJ ( S1, p, j, q )
? ⇐ RES ( jn )
```

C.6.32
```
RES ( p1 ) ⇐ P ( p, pn, p1, w, pc ) AND
             SPJ ( S1, p, j, q )
? ⇐ RES ( p1 )
```

C.6.33
```
RES ( p ) ⇐ P ( p, pn, p1, w, pc ) AND
            SPJ ( s, p, j, q ) AND
            J ( j, jn, London )
? ⇐ RES ( p )
```

C.6.34
```
RES ( j ) ⇐ SPJ ( s, p, j, q ) AND
            SPJ ( S1, p, j2, q2 )
? ⇐ RES ( j )
```

C.6.35
```
RES ( s ) ⇐ SPJ ( s, p, j, q ) AND
            SPJ ( s2, p, j2, q2 ) AND
            SPJ ( s2, p2, j3, q3 ) AND
            P ( p2, pn, Red, w, c )
? ⇐ RES ( s )
```

C.6.36
```
RES ( s ) ⇐ S ( s, sn, st, c ) AND
            S ( S1, sn1, st1, c1 ) AND st < st1
? ⇐ RES ( s )
```

C.6.37–C.6.39 Cannot be done without grouping and aggregate functions.

C.6.40–C.6.44 Cannot be done without negation.

C.6.45
```
RES ( c ) ⇐ S ( s, sn, st, c )
RES ( c ) ⇐ P ( p, pn, p1, w, c )
```

```
RES ( c ) ⇐ J ( j, jn, c )
? ⇐ RES ( c )
```

C.6.46
```
RES ( p ) ⇐ SPJ ( s, p, j, q ) AND
            S ( s, sn, st, London )
RES ( p ) ⇐ SPJ ( s, p, j, q ) AND
            J ( j, jn, London )
? RES ( p )
```

C.4.47–C.4.48 Cannot be done without negation.

C.7 We show the constraints as conventional implications instead of in the "backward" Datalog style. We have numbered them C.7.*n*, where 16.*n* is the number of the original exercise in Chapter 16.

C.7.1
```
CITY ( London )
CITY ( Paris )
CITY ( Rome )
CITY ( Athens )
CITY ( Oslo )
CITY ( Stockholm )
CITY ( Madrid )
CITY ( Amsterdam )

S ( s, sn, st, c )      ⇒ CITY ( c )
P ( p, pn, pc, pw, c ) ⇒ CITY ( c )
J ( j, jn, c )          ⇒ CITY ( c )
```

C.7.2
```
P ( p, pn, Red, pw, pc ) ⇒ pw < 50
```

C.7.3 Cannot be done without negation.

C.7.4
```
S ( s1, sn1, st1, Athens ) AND
S ( s2, sn2, st2, Athens ) ⇒ s1 = s2
```

C.7.5–C.7.6 Cannot be done without grouping and aggregate functions.

C.7.7
```
J ( j, jn, c ) ⇒ S ( s, sn, st, c )
```

C.7.8
```
J ( j, jn, c ) ⇒ SPJ ( s, p, j, q ) AND
                  S ( s, sn, st, c )
```

C.7.9
```
P ( p1, pn1, pl1, pw1, pc1 ) ⇒
P ( p2, pn2, Red, pw2, pc2 )
```

C.7.10 Cannot be done without grouping and aggregate functions.

C.7.11 Cannot be done without negation.

C.7.12
```
P ( p1, pn1, pl1, pw1, pc1 ) ⇒
P ( p2, pn2, Red, pw2, pc2 ) AND pw2 < 50
```

C.7.13 Cannot be done (this is a transition constraint).

C.7.14–C.7.15 Cannot be done without grouping and aggregate functions.

C.7.16 Cannot be done (this is a transition constraint).

APPENDIX D

Abbreviations and Acronyms

ACID	atomicity/consistency/isolation/durability
ACM	Association for Computing Machinery
ADT	abstract data type
AK	alternate key
ANSI	American National Standards Institute
ANSI/SPARC	literally, ANSI/Systems Planning and Requirements Committee; used to refer to the three-level database system architecture described in Chapter 2
BCNF	Boyce/Codd normal form
BCS	British Computer Society
BLOB	basic (*or* binary) large object
BNF	Backus/Naur form *or* Backus normal form
CACM	Communications of the ACM (ACM publication)
CAD/CAM	computer-aided design/computer-aided manufacturing
CASE	computer-aided software engineering
CDO	class-defining object
CIM	computer-integrated manufacturing
CK	candidate key
CODASYL	literally, Conference on Data Systems Languages; used to refer to certain prerelational (network) systems such as IDMS
CPU	central processing unit
CS	cursor stability (DB2)
DA	data administrator
DB/DC	database/data communications
DBA	database administrator
DBMS	database management system
DBRM	database request module (DB2)
DBTG	literally, Data Base Task Group; used interchangeably with CODASYL

DB2	IBM DATABASE 2
DC	data communications
DDB	distributed database
DDBMS	distributed DBMS
DDL	data definition language
DES	Data Encryption Standard
DK/NF	domain-key normal form
DL/I	Data Language/I (IMS)
DML	data manipulation language
DRDA	Distributed Relational Database Architecture (IBM)
DSL	data sublanguage
DUW	distributed unit of work
E/R	entity/relationship
EDB	extensional database
EKNF	elementary key normal form
EMVD	embedded MVD
FD	functional dependence
FK	foreign key
I/O	input/output
IDB	intensional database
IDMS	Integrated Database Management System
IMS	Information Management System
IND	inclusion dependence
INGRES	Interactive Graphics and Retrieval System
INGRES/STAR	distributed version of INGRES
IRDS	Information Resource Dictionary Systems
IS	intent shared (lock), *also* information systems
ISO	International Organization for Standardization
IT	information technology
IX	intent exclusive (lock)
JD	join dependence
LAN	local area network
MLS	multilevel secure
MVD	multivalued dependence
NF^2	"NF squared" = non1NF
OID	object ID
OO	object-oriented *or* object-orientation
OODB	object-oriented database
OOPL	object-oriented programming language

OSI	Open Systems Interconnection
OSQL	Object SQL
PJ/NF	projection-join normal form
PK	primary key
PRTV	Peterlee Relational Test Vehicle
QBE	Query-By-Example
QMF	Query Management Facility
QUEL	Query Language (INGRES)
R*	"R star" (distributed version of System R)
RAID	redundant array of inexpensive disks
RDA	Remote Data Access
RDB	relational database
RDBMS	relational DBMS
RID	(stored) record ID
RM/T	relational model/Tasmania
RM/V1	relational model/Version 1
RM/V2	relational model/Version 2
RPC	remote procedure call
RR	repeatable read (DB2)
RUW	remote unit of work
RVA	relation-valued attribute
S	shared (lock)
SAG	SQL Access Group
SDD-1	System for Distributed Databases - 1
SIGMOD	Special Interest Group on Management of Data (ACM special interest group)
SIX	shared intent exclusive (lock)
SPARC	*see* ANSI/SPARC
SQL	(originally) Structured Query Language
SQL/DS	SQL/Data System
SQL/92	the current ISO/ANSI SQL standard
TCB	Trusted Computing Base
TCP/IP	Transmission Control Protocol/Internet Protocol
TID	(stored) tuple ID
TODS	Transactions on Database Systems (ACM publication)
U	update (lock)
unk	*unknown* (truth value)
UNK	unknown (null)
UOW	unit of work

VLDB	Very Large Data Bases (annual conference)
VSAM	Virtual Storage Access Method (IBM)
WAL	write-ahead log
WAN	wide area network
WFF	well-formed formula
WORM	write-once/read-many-times
WYSIWYG	what you see is what you get
X	exclusive (lock)
X3H2	ANSI database committee
1NF	first normal form
2NF	second normal form
2PC	two-phase commit
2PL	two-phase locking
2VL	two-valued logic
3GL	third generation language
3VL	three-valued logic
3NF	third normal form
4GL	fourth generation language
4NF	fourth normal form
4VL	four-valued logic
5NF	fifth normal form (same as PJ/NF)

Index

Abiteboul, Serge, 684, 814
Abrial, Jean Raymond, 366
abstract data type
  see data type, domain
abstract syntax tree
  see query tree
access methods, 712ff
access path selection, 507
ACID properties, 379
active domain, 102
Adachi, S., 531
Adiba, Michel E., 494, 626
Adleman, L., 428, 437
adopting a premise, 781
agent, 604
aggregate function
  relational algebra, 162
  relational calculus, 201
  SQL, 229
Agrawal, Rakesh, 565, 810, 811, 814
Aho, Alfred V., 330, 339, 812
alerter
  see triggered procedure
ALL (SQL)
  see duplicates
all-or-any condition (SQL), 242
Allen, Frank W., 366
ALPHA
  see DSL ALPHA
ALTER DOMAIN (SQL), 222
ALTER TABLE (SQL), 89, 224
alternate key, 115
Altman, Edward B., 749
American National Standards Institute
  see ANSI
anchor point, 735
Andersen, Erika, 568
ANSI, 65
ANSI/SPARC, 28, 50
ANSI/SPARC architecture, 28ff
  vs. SQL, 72
ANSI/X3, 50
ANSI/X3/SPARC Study Group on Data Base Management Systems
  see ANSI/SPARC
Anton, Jeff, 67
APPEND (QUEL), 424
application plan (DB2), 764, 766
application preparation (DB2), 764ff
application programmer, 7
ARIES, 389
arity
  see degree
Armstrong, W. W., 275, 281, 282, 339, 340
Armstrong's axioms, 275, 282
Arya, Manish, 565
Ashenhurst, Robert L., 73
assertion (SQL)
  see CREATE ASSERTION
assignment (relational)
  see relational assignment
association (RM/T), 367
associative entity (RM/T)
  see association (RM/T)
associativity, 149
Astrahan, Morton M., 255, 256, 749
Atkinson, Malcolm P., 684, 704
atomicity
  relations, 305
  scalar values, 55, 93, 547ff
  transactions, 376, 379
attribute, 79, 87
  relation-valued 560ff
attribute integrity (metarule), 130

audit trail, 422
authentication
  see password
authority
  see privilege
authorization
  see security
automatic navigation
  see navigation
availability, 598
AVG
  see aggregate function
axiom (database), 776

B-tree, 729ff
backend, 42
backward chaining, 781
Badal, B. Z., 457
Bancilhon, François, 704, 810, 811, 812
Banerjee, J., 684
Barnes, G. Michael, 749
Barnsley, Michael F., 757
base relation
  see relation
base table, 62
Batory, Don S., 526, 750
Bayer, Rudolf, 410, 729, 750
BCNF, 290, 306, 313
Beckley, D. A., 756
Beech, David, 705
Beeri, Catriel, 330, 339, 812
BEGIN DECLARE SECTION (SQL), 246
BEGIN TRANSACTION, 377
Bell, David, 623
Bentley, J. L., 756
Bernstein, Philip A., 315, 386, 411, 457, 495, 623, 624
bill of materials, 12, 208, 212
  see also parts explosion, recursive queries
binary relation
  see degree
Bind (DB2), 764, 766
Bitton, Dina, 528, 529
Bjork, L. A., 386
Björnerstedt, Anders, 684
Blasgen, Michael W., 256, 411, 527
Blaustein, Barbara T., 457
block
  see page
body (relation), 87
Bonner, Anthony, 684
Booch, Grady, 649
Boral, Haran, 529
bound variable, 189, 192
Boyce, Raymond F., 256
Boyce/Codd normal form
  see BCNF
Breitbart, Yuri, 624
Brodie, Michael L., 809
Brosda, Volkert, 340
brute force
  see join implementation
bucket, 751
Buff, H. W., 494
Buffer Manager (DB2), 766, 767
Buneman, O. Peter, 458, 684

C++, 684
caching, 674
calculus, 778
candidate key, 112ff, 452
  and nulls, 124
  inheritance, *see* inheritance
  SQL, 223
Cannan, Stephen, 255
canonical form, 505
cardinality, 79, 87
Carlson, C. Robert, 340
Cartesian product, 141, 147
  extended, 147
Casanova, Marco A., 283, 495
CASCADE (SQL), 222, 224, 225, 433, 454
CASCADES
  see foreign key rules
CASE (SQL), 244
Casey, R. G., 283
CAST (SQL), 244
catalog, 60, 74, 106
  distributed, 608
  OO, 681
  SQL, *see* Information Schema
Cattell, R. G. G., 649, 674, 684, 705
Ceri, Stefano, 458, 624, 814
Chamberlin, Donald D., 255, 256, 458, 495, 565, 773
Chang, C. C., 627
Chang, C. L., 208
Chang, Ellis, 684
Chang, H. K., 751
Chang, Philip Yen-Tang, 530
characteristic (RM/T), 367
characteristic entity (RM/T)
  see characteristic (RM/T)

chase, 339
CHECK option (SQL), 490
checking time, 440
checkpoint, 380
Chen, Peter Pin-Shan, 348, 351, 355, 362, 366, 367, 370
Cheng, Josephine M., 535
Chiang, T. C., 754
Chignell, Mark, 650
Christodoulakis, Stavros, 756
chronological sequence, 723
ciphertext, 427
Clarke, Edmund M., 457
class
  see object class
class-defining object, 636
class hierarchy
  see type hierarchy
classification level, 425
clausal form, 782
clearance level, 425
Cleaveland, J. Craig, 565
Clemons, Eric K., 458
client
  see client/server
client/server, 42ff, 616ff
  SQL, 618
CLOSE cursor (SQL), 250
closed WFF, 784
Closed World Assumption
  see CWA
closure
  attributes, 277
  FDs, 275
  relational algebra, 54, 141ff, 467, 513
clustering, 674, 716
  inter-file, 717
  intra-file, 716
clustering index
  DB2, 763
  see also index
Codd, Edgar F., *passim*
Codd's reduction algorithm, 197ff
coercion, 545
collection object, 643
collision, 735
collision chain, 735
combined index
  see index
Comer, Douglas, 750
commalist (BNF), 91
command-driven interface, 8
comment (SQL), 228
COMMIT, 376
  SQL, 384
commit point, 377
commutativity, 149, 510
compensating transaction, 387
compilation (DB2), 767
complex object
  see object
composite domain
  see domain
compression, 740
  differential, 742
  fractal, 757
  front, 742
  hierarchic, 743
  rear, 742
conceptual database design
  see database design
conceptual DDL, 35
conceptual record, 34
conceptual schema, 35
conceptual view, 34
conceptual/internal mapping, 36
conclusion (proof), 780
concurrency, 391ff
  distributed, 612
conditional expression (SQL), 241ff
configuration, 671
conjunct, 511
conjunctive normal form, 511, 781
CONNECT (SQL), 618
connection trap, 12, 332, 346
consistency (transaction), 379
  see also inconsistency, integrity
constraint
  see integrity constraint
constructor function, 549
containment hierarchy, 638
controlled redundancy
  see redundancy
convoy, 411
Coordinator, 383
Copeland, George, 667
cost formula, 506
COUNT
  see aggregate function
cover (FDs), 278
  irreducible, *see* irreducibility
Cox, Brad J., 649
CREATE ASSERTION (SQL), 454
CREATE BASE RELATION, 84, 90–91

CREATE DOMAIN, 83, 548, 551
  SQL, 221
CREATE INTEGRITY RULE, 443
CREATE SECURITY RULE, 419
CREATE SNAPSHOT, 489
CREATE TABLE (SQL), 65, 223
CREATE VIEW, 62, 470
  SQL, 68, 490
criterion for update acceptability, 97, 168
Crus, Richard A., 386
CS
  see isolation level
cursor (SQL), 248
cursor stability
  see isolation level
CWA, 792

Dahl, O. J., 649
Dahl, Veronica, 810
Daniels, D., 624
DAPLEX, 706
Dar, Shaul, 814
Darwen, Hugh A. C., 103, 104, 132, 133, 142, 174, 175, 209, 255, 276, 281, 284, 495, 565, 589
Das, Subrata Kumar, 814
data administrator, 13
data communications, 41
data communications manager
  see DC manager
data compression
  see compression
data definition language
  see DDL
data dictionary
  see dictionary
data encryption
  see encryption
Data Encryption Standard
  see DES
data fragmentation
  see fragmentation
data independence, 16ff
  DB2, 768
  logical, 468
  OO, 636
  physical, 468
  see also encapsulation
Data Independent Accessing Model
  see DIAM
data integration, 6
data manipulation language
  see DML
data model, 103, 348
data protection
  see concurrency, integrity, recovery, security
data replication
  see replication
data repository
  see dictionary
data sharing, 6
data sublanguage
  see DSL
Data Sublanguage ALPHA
  see DSL ALPHA
data type, 543ff
  abstract, 543
  SQL, 221
  user-defined, 543
  see also domain
database, 2, 9ff
  advantages, 13–14
  DB2, 762
database/data communications system
  see DB/DC system
database administrator
  see DBA
database design, 268ff
  conceptual, 37
  logical, 37
  physical, 37
database management system
  see DBMS
database manager
  see DBMS
database predicate, 450
database procedure, 122
  see also stored procedure, triggered procedure
database programming languages, 669ff
Database Request Module (DB2)
  see DBRM (DB2)
database statistics, 513
  DB2, 513–514
  INGRES, 514
database system, 2, 4ff
Datalog, 796ff
Date, C. J., *passim*
Davies, C. T., Jr., 387, 390
Dayal, Umeshwar, 495
DB manager
  see DBMS
DB/DC system, 42
DBA, 14, 37ff
DBMS, 7, 39ff

DBMS independence, 605
DBRM (DB2), 764, 766
DB2, 132, 133, 384, 386, 388, 403, 500, 507, 526, 535, 592, 761ff
  components, 764
DB2 Distributed Data Facility, 896
DC manager, 41, 383
DDBMS, 595
DDL, 33
de Maindreville, Christophe, 813
de Maine, P. A. D., 753
De Morgan's Laws, 778
deadlock, 398
  global, 613
DECLARE CURSOR (SQL), 249
decryption
  see encryption
deductive axiom, 777, 793, 802
deductive database, 807
deductive DBMS, 793
DEE
  see TABLE_DEE and TABLE_DUM
default values, 128, 591
  SQL, 221, 223
deferred checking (SQL), 455
DEFINE INTEGRITY (QUEL), 459
DEFINE PERMIT (QUEL), 424
degree, 79, 87, 89
  E/R, 353
degree of consistency
  see isolation level
DELETE, 167
  CURRENT (SQL), 250
  embedded (SQL), 248
  SQL, 67, 235
delete rule
  see foreign key rules
Delobel, Claude, 283, 340, 344
DeMichiel, Linda, 565
Denning, Dorothy E., 435
Denning, Peter J., 435
dense index
  see index
dependency
  see FD, IND, JD, MVD
dependency diagram
  see FD diagram
dependency preservation
  see FD preservation
dependency theory, 338
dependent (FD), 273
derived relation
  see relation
derived table, 62
DES, 428, 435
descriptor, 61
designation (RM/T), 367
designative entity (RM/T)
  see designation (RM/T)
DESTROY BASE RELATION, 91
DESTROY DOMAIN, 84, 551
DESTROY SECURITY RULE, 420
DESTROY SNAPSHOT, 489
DESTROY VIEW, 471
detachment
  see query decomposition
DETECT (OPAL), 663
determinant (FD), 273
DeWitt, David J., 528, 529
DIAM, 749
dictionary, 40, 365
  see also catalog
difference, 141, 146
  SQL, *see* EXCEPT
differential compression
  see compression
differential file, 754
Diffie, W., 428, 436
digital search tree
  see trie
digital signature, 430
direct access (index), 725
direct hash
  see hashing
direct invocation (SQL), 68
dirty read, 407
DISCONNECT (SQL), 618
discretionary access control, 417ff
disjunctive information, 792
disk directory, 720
disk manager, 713–714
disk table of contents
  see disk directory
DISTINCT (SQL), 227
  aggregate functions, 229
distributed database, 48, 593ff
  fundamental principle, 596
distributed DBMS
  see DDBMS
Distributed INGRES, 596, 624ff
distributed processing, 44–45
Distributed Relational Database Architecture
  see DRDA
distributed request (DRDA), 623
distributed unit of work (DRDA), 623

distribution independence, 621
distributivity, 778
divide, 141, 155
  see see also generalized divide
DIVIDEBY
  see divide
DIVIDEBY PER, 176
division/remainder (hash), 733
DK/NK, 337
DML, 33
Dodd, M., 751
domain, 79, 81ff, 543ff
  as data type, 85
  composite, 550
  relation-valued, 560ff
  simple, 550
  SQL, 220
domain calculus, 186, 203ff
domain check override, 544
domain variable, 186, 203
domain-key normal form
  see DK/NF
double index
  see index
DRDA, 619, 622
DROP ASSERTION (SQL), 454
DROP DOMAIN (SQL), 222
DROP TABLE (SQL), 225
DROP VIEW (SQL), 491
DSL, 33
  loosely coupled, 33
  tightly coupled, 33
DSL ALPHA, 185, 212
dual-mode principle, 244
DUM
  see TABLE_DEE and TABLE_DUM
dump, 382
dump/restore, 382
  see also unload/reload
duplicates, 92, 104
  SQL, 222, 227, 234
  and three-valued logic, 575ff
durability (transaction), 379
dynamic hashing
  see hashing
dynamic SQL, 250ff

E-relation (RM/T), 349
E/R diagram 10–11, 355ff
E/R model, 347ff
EDB, 792
elementary key normal form, 316
Ellis, Margaret A., 684
Elmasri, Ramez, 368, 749
embedded MVD
  see EMVD
embedded SQL, 68, 244ff
  and three-valued logic, 585
EMVD, 341
encapsulation, 548, 630, 635, 690
Encarnaçao, J., 756
encryption, 426
  public-key, 428ff
encryption key, 427
end user, 8
entity, 10, 349, 350, 351
  vs. relationship, 12, 69, 363
entity classification (RM/T), 367
entity integrity (metarule), 124
entity subtype
  see subtype
entity supertype
  see subtype
entity/relationship diagram
  see E/R diagram
entity/relationship model
  see E/R model
Epstein, Robert S., 536, 624
equijoin, 154
escalation
  see lock escalation
escrow locking, 413
Eswaran, Kapali P., 412, 458, 459, 527
Evens, M. W., 756
EXCEPT (SQL), 234
exclusive lock
  see X lock
EXEC SQL, 245
EXECUTE (SQL), 251
existential quantifier
  see EXISTS
EXISTS, 190
  SQL, 232
  three-valued logic, 574
expert database, 807
expert DBMS, 807
EXPLAIN (DB2), 770
explicit dynamic variable, 644
extend, 160
extendable hashing
  see hashing
extendable system, 708
extended Cartesian product
  see Cartesian product

extensional database
  see EDB
external DDL, 34
external record, 34
external schema, 34
external view, 34
external/conceptual mapping, 36
external/external mapping, 37

Fagin, Ronald, 283, 284, 327, 328, 329, 332, 333, 334, 335, 339, 340, 341, 342, 344, 436, 752
FD, 271ff
  axiomatization, *see* Armstrong's axioms
  transitive, 275
  trivial, 274
FD diagram, 294
FD preservation, 303ff
federated system, 624
Fehder, Paul L., 749
Fernandez, Eduardo B., 435, 436
FETCH, 93
  SQL, 250
fifth normal form, 333
file manager, 713, 715
  vs. DBMS, 41
final normal form
  see fifth normal form
Finkel, R. A., 756
Finkelstein, Sheldon, 532, 755
FIPS, 255
first normal form, 93, 289, 296
fixpoint, 800, 812
Fleming, Candace C., 368
flow controls, 435
FORALL, 190
  three-valued logic, 574
Ford, Daniel Alexander, 756
foreign key, 116ff, 452
  and nulls, 126
  SQL, 223
foreign key rules, 120ff, 127–128
forms-driven interface, 8
formula, 779
  see also WFF
forward chaining, 780
four-valued logic, 571
fourth generation language, 31
fourth normal form, 329
fragmentation, 599
fragmentation independence, 601
fragmentation transparency
  see fragmentation independence
Franaszek, Peter A., 412
Fredkin, E., 749, 754
free space page set
  see page set
free variable
  see bound variable
Freeston, Michael, 757
Freytag, Johann Christoph, 540, 709
front compression
  see compression
frontend, 42
Fry, James P., 354, 371, 748
full FD, 294
fully inverted, 727
functional dependency
  see FD
functionally determines
  see FD
Furtado, Antonio L., 177, 495
further normalization, 288ff
  not a panacea, 336
  reversibility, 290

Gallaire, Hervé, 809, 810
Ganski, Richard A., 536, 537
Garcia-Molina, Hector, 387, 624
Gardarin, Georges, 459, 798, 810, 812, 813
Garg, Anil K., 753
gateway, 614ff
GEM, 707
GemStone, 649, 652
generalized divide, 175
generalized summarize, 174
GET_*v*, 659
Ghosh, Sakti P., 749
Gilbert, Arthur M., 256
global deadlock
  see deadlock
Goldberg, Adele, 649
Goldstein, R. C., 176
Goodman, Nathan, 386, 411, 495, 540, 625, 649, 705
Gotlieb, C. C., 753
Gottlob, G., 814
Graefe, Goetz, 541
GRANT (SQL), 432
grant authority (SQL), 432
Grant, John, 533, 625
GRANT option (SQL), 432
granularity of locks
  see lock granularity

Gray, James N., 387, 388, 411, 412, 413, 495, 625
Gray, Peter M. D., 809
greater-than join, 154
Griffiths, Patricia P.
  see Selinger, Patricia G.
Grimson, Jane, 623
ground axiom, 776, 789
GROUP BY (SQL), 230, 238
grouped table (SQL), 238
growth in the database, 468
Grumbach, Stéphane, 814
Gupta, Anoop, 539
Gupta, Gopal K., 751
Guttman, Antonin, 756

Haas, Laura M., 709
Hackathorn, Richard D., 625
Haderle, Don, 389
Hadzilacos, Vassos, 386
Hall, Patrick A. V., 142, 177, 368, 530
Hammer, Michael M., 369, 459, 625
Han, Jiawei, 813
Hanson, Eric, 567, 568
Harary, Frank, 749
hard crash, 380
Härder, Theo, 388, 413
Hasan, Waqar, 531
hash-addressing
  see hashing
hash-indexing
  see hashing
hashing, 733ff
  direct, 733
  dynamic, 752
  extendable, 735ff
  indirect, 733
  linear, 752
  order-preserving, 753
  perfect, 753
  trie, 752
  virtual, 752
hash join
  see join implementation
hash lookup
  see join implementation
HAVING (SQL), 230, 238
heading (relation), 87
heap, 747
Heath, Ian J., 294, 316, 591
Heath's theorem, 294, 317, 329
Heaven, T. E., 566
Held, Gerald D., 209, 210
Heller, M., 410
Hellerstein, Joseph M., 531
Hellman, M. E., 428, 436
Henschen, Lawrence J., 813
hidden data, 470
hierarchic compression
  see compression
hierarchic system, 23
Hinterberger, H., 756
Hitchcock, Peter, 177
Horn clause, 787
Horowitz, Bruce M., 133
host language, 33
hotspot, 413
Howard, J. H., 339
Hsiao, David K., 749
Huffman, D. A., 753
Huffman coding, 745
Hull, Richard, 369
Hultén, Christer, 684

IDB, 793
idempotence, 50
identity
  multiplication, 171
  relational algebra, *see* TABLE_DEE and TABLE_DUM
IF . . . THEN . . .
  see logical implication
IF_UNK, 572
impedance mismatch, 670
implication
  see logical implication
implication problem, 339, 342
IN, 170
  SQL, 231
inclusion dependency
  see IND
inconsistency, 14
inconsistent analysis, 394, 398
IND, 283
independent projections, 304
index, 725
  clustering, 757
  combined, 727
  dense, 728
  double, 747
  multilevel, 729
  nondense, 729
  primary, 725
  secondary, 725

sparse, 728
tree-structured, 729
unique, 725
see also B-tree
index lookup
see join implementation
index set (B-tree), 730
index space (DB2), 762
indicator variable (SQL), 585
indirect hash
see hashing
inference controls, 435
inferential DBMS, 807
inflight checking, 132
Information Resource Dictionary System
see IRDS
Information Schema (SQL), 68, 225
INGRES, 133, 210, 388, 423, 437, 459, 536, 568
INGRES/STAR, 596
inheritance
attribute name, 142ff
behavioral, 645
candidate key, 143, 284
domains, 554
FD, 284
integrity constraint, 441
multiple, 559, 647
relations, 557
structural, 645
type, *see* type hierarchy
INSERT, 167
embedded (SQL), 248
SQL, 67, 234
instance variable, 636
private, 637
protected, 637
public, 637
Instrumentation Facility (DB2), 771
integrated
see data integration
integrity, 416, 440ff
OO, 677
SQL, 454ff
see also candidate key, dependency, foreign key, integrity constraint, integrity rules, metarules
integrity constraint, 443
immediate *vs.* deferred, 446, 449
immediate *vs.* deferred (SQL), 455
state *vs.* transition, 451
integrity rule classification scheme, 444
integrity rules, 442ff
attribute, 445
classification scheme, 444
database, 449
database-specific, 111, 441
domain, 444
relation, 446
see also metarules
intensional database
see IDB
intent locking, 404, 412
inter-file clustering
see clustering
internal DDL, 35
internal record, 35
see also stored record
internal schema, 36
see also storage structure
internal view, 35
International Organization for Standardization
see ISO
interpretation (logic), 784
intersect, 141, 146
SQL, 234
intra-file clustering
see clustering
inverse variable, 676
inverted list
see index
inverted list system, 22–23
Ioannidis, Yannis E., 538, 540, 811
IRDS, 51
irreducibility
candidate key, 114
cover, 280
FD, 278, 294
is deducible from ($\vdash$), 780
IS/1, 177
ISBL, 177
ISO, 50, 219, 255, 366, 622
isolation level, 402ff
CS (DB2), 403
RR (DB2), 403
SQL, 384, 406–407
isolation (translation), 379
IS_UNK, 572
it is always the case that ($\models$), 780

Jaeschke, G., 565
Jagadish, H. V., 684, 811, 814
Jagannathan, D., 369

Jajadia, Sushi, 436
Jarke, Matthias, 527, 531
JD, 332
  axiomatization, 344
join, 53, 141, 152
  see also equijoin, natural join, outer join, θ-join
join dependency
  see JD
join implementation, 519ff
journal
  see log

Kaplan, Robert S., 340
Katz, Randy H., 684
KBMS, 807
Keller, Arthur M., 495
Kemnitz, Greg, 708
Kent, William, 51, 316, 342, 343, 366, 369
kernel (RM/T), 367
kernel entity (RM/T)
  see kernel (RM/T)
Kerschberg, Larry, 177, 809
key constraint (DK/NF)
  see DK/NF
key transformation
  see hashing
Khoshafian, Setrag, 650
Kiessling, Werner, 536, 537
Kifer, Michael, 705, 814
Kim, Won, 526, 536, 537, 538, 650, 705
King, Jonathan J., 533
King, Roger, 369, 650
Kitakami, H., 531
Klug, Anthony, 50, 178
Knott, G. D., 752
knowledge, 806
knowledge base, 806
knowledge base management system
  see KBMS
Knuth, Donald E., 730, 748
Koch, Jürgen, 527, 531
Korth, Henry F., 343, 388, 566, 749
Krause, F. -L., 756
Kreps, Peter, 210
Krishnamurthy, Ravi, 540
Kuhns, J. L. , 186, 210
Kung, H. T., 413

Lacroix, M., 186, 210, 591
Larson, Per-Åke, 752, 753
Lavazza, L., 813
Lavender Book, 425, 435
left-irreducible (FD), 278, 294
Leifker, Daniel, 756
Lempel, Abraham, 437
Levy, Eliezer, 388
Lewis, T. G., 751
Lindsay, Bruce G., 389, 494, 565, 625, 708
linear hashing
  see hashing
list (BNF), 91
Litwin, Witold, 652, 752
Liu, Ken-Chih, 591
load, 44
local autonomy, 598, 752
location independence, 599
location transparency
  see location independence
Lochovsky, Fred H., 650
lock escalation, 406
lock granularity, 404
locking, 395ff
log, 377
  active, 377
  archive, 377
logic as a data model, 807
logic-based system, 775ff, 807
logic database, 807
logical data independence
  see data independence
logical database design
  see database design
logical implication, 196
logical theory, 776, 791
Lohman, Guy M., 539, 540, 709, 754
Lomet, David B., 753, 756
long transaction, 672
Loomis, Mary E. S., 366
Loosley, Chris R., 535
Lorie, Raymond A., 255, 389, 412, 413, 773
lossless decomposition
  see nonloss decomposition
lost update, 392, 396
Lozinskii, Eliezer L., 812, 814
Lu, Hongjun, 811
Lucchesi, Claudio L., 284
Lum, Vincent Y., 751
Lyngbaek, Peter, 706

MacAIMS, 176
magic, 532, 811
Maier, David, 343, 591, 650, 667, 811

Makinouchi, A., 531, 566
mandatory access control, 417, 425–426
Manna, Zohar, 809
Mannila, Heikki, 369
Mannino, Michael V., 366
manual navigation
  see navigation
Markowitz, Victor M., 133
Marron, B. A., 753
Martin, James, 748
Maryanski, Fred, 370
MATCH (SQL), 241
MATCHING, 163, 170
materialization (view processing), 472
Maurer, W., 751
MAX
  see aggregate function
MAYBE, 573
McCord, Michael, 809
McCreight, C., 729, 750
McGoveran, David, 440, 496, 591
McLeod, Dennis J., 105, 369
McPherson, John, 708
media failure, 380, 382
Melkanoff, Michel, 459
Melton, Jim, 255
membership condition (domain calculus), 204
Mendelzon, Albert O., 342
menu-driven interface, 8
Merrett, T. H., 177, 528, 748
message, 635
metarules, 129
method, 635
MIN
  see aggregate function
minimality
  see irreducibility
Minker, Jack, 533, 809, 810
MINUS
  see difference
missing information, 570ff
  OO, 681
  see also null, three-valued logic
Mitama, M., 411
model (logic), 785, 791
model-theoretic, 776
modus ponens, 780
Mohan, C., 389, 625
Morris, R., 751
Mullin, James K., 751
multidatabase system, 624
multidependency
  see MVD
multidetermines
  see MVD
multilevel index
  see index
multilevel secure system, 426, 436, 437
multiuser system, 5
multivalued dependency
  see MVD
multiversion locking, 410, 672
Mumick, Inderpal Singh, 532
Muralikrishnan, M., 536, 538
MVD, 328
  axiomatization, 339
  embedded, *see* EMVD
Myhrhaug, B., 649
Mylopoulos, John, 809

n-ary relation
  see relation
n-decomposable, 330
n-tuple
  see tuple
naive evaluation, 800
Nakano, Ryohei, 539
named relation
  see relation
Naqvi, Shamim, 814
natural join, 141, 152
Naughton, Jeffrey F., 813
Navathe, Shamkant B., 368, 749, 813
navigation, 58
Negri, M., 257, 528
Nejdl, Wolfgang, 813
Ness, L., 811
NEST, 561, 567
nested relation, 560
nested transaction, 390, 672
  see also savepoint
network system, 23
Neuhold, Erich J., 627
NEW, 642
Newman, Scott, 625
NF^2 relation, 560
Ng, Raymond T., 540
Nicolas, Jean-Marie, 330, 343, 810
Nierstrasz, O. M., 650
Nievergelt, Jürg, 752, 756
Nilsson, J. F., 773
NO ACTION (SQL)
  see CASCADE

nondense index
  see index
nonloss decompositon, 292ff
nonrepeatable read, 407
normal form, 289
normalization, 94
  see also further normalization
normalization procedure, 290, 335
normalized relation, 93
Notley, M. Garth, 177
null, 123, 570ff
  and candidate keys, 124
  and foreign keys, 126
  SQL, 582ff
  see also missing information, three-valued logic
nullary projection
  see project
nullary relation
  see TABLE_DEE and TABLE_DUM
NULLIFIES
  see foreign key rules
nullology, 104
nulls rule
  see foreign key rules
Nygaard, K., 649

O'Neil, Patrick E., 413
object, 635, 650
object class, 635, 642
  vs. domain, 550, 690ff
  vs. relation, 693ff
object ID
  see OID
object instance
  see object
object oriented
  see OO
Object Oriented Database System Manifesto, 704
"Object SQL"
  see OSQL
OID 637, 642, 651
Olle, T. William, 370
online application, 8
OO, 629ff
  good ideas of, 689
OO1 benchmark, 670
OODBMS, 629ff
  as a construction kit, 681
OPAL, 649, 652ff
OPEN cursor (SQL), 249
open WFF
  see closed WFF
optical disk, 756
optimistic concurrency control, 413
optimizability, 502
optimization, 58, 159, 501ff
  DB2, 766
  distributed, 603, 606ff
  semantic, 512
  three-valued logic, 577
optimization inhibitors, 524
OPTIMIZEDB (INGRES), 514
Orange Book, 425, 435
ORACLE7 distributed database option, 596
ORDER BY (SQL), 227
ordering
  column (attribute), 93
  column (SQL), 223
  row (tuple), 92
  row (SQL), 227
orthogonality, 230
Osborn, Sylvia L., 284, 344, 566
OSQL, 705–706
Otten, Gerard, 255
outer difference, 582
outer intersection, 582
outer join, 579ff, 589
  SQL, 583
outer union, 581, 582
  SQL, 583
overflow page, 723, 735
overloading
  see polymorphism
Owlett, J., 368
Ozsoyoglu, Z. Meral, 533
Özsu, M. Tamer, 625

P-relation (RM/T), 349
page, 36, 714ff
  DB2, 763
page header, 718
page management, 717
page set, 714
  free space, 715
page zero
  see disk directory
Palermo, Frank P., 529
Papidimitriou, Christos H., 283, 414
parent/child storage structure, 738
Parker, D. S., 340, 344
Parsaye, Kamran, 650
participant

E/R, 353
two-phase commit, 383
parts explosion, 265
see also bill of materials, recursive queries
password, 418
path expression, 666
Pecherer, Robert M., 527
Pelagatti, Giuseppe, 257, 528, 624,
Performance Monitor (DB2), 771
phantom, 407
physical database design
see database design
physical record
see page
physical sequence, 723
Pippenger, Nicholas, 752
Pirahesh, Hamid, 389, 531, 532, 708, 709
Pirotte, Alain, 186, 210, 591
PJ/NF
see fifth normal form
plaintext, 426
planned request, 40
pointer chain, 738
polymorphism, 557, 645
Popek, G. J., 457
POSTGRES, 708
Precompiler (DB2), 764, 766
predicate, 97, 168, 783
database, *see* database predicate
derived relation, 168
relation, *see* relation
predicate calculus, 782ff
"predicate transitive closure", 511
prefix
see record prefix
premise (proof), 780
premiss
see premise
prenex normal form, 194, 786
PREPARE (SQL), 251
Price, T. G., 411
primary copy, 610, 613
primary domain, 131
primary index
see index
primary key, 79, 115
primitive operators
see relational algebra
private memory, 636
privilege, 418
SQL, 432
product
see Cartesian product
project, 53, 141, 151
identity projection, 151
nullary projection, 151
projection-join normal form
see fifth normal form
proof, 780
proof-theoretic, 776, 789, 791
propagating updates
see update propagation
property, 12, 349, 350, 353, 367
PROPERTY (RM/T), 349
proposition, 97
propositional calculus, 778ff
PRTV, 177, 531
public interface, 636
public-key encryption
see encryption
Putzolu, Gianfranco R., 412, 413

QBE, 186, 210
Qian, Xiaolei, 684
qualified name (SQL), 227
quantifier, 190
see also EXISTS, FORALL
QUEL, 186, 210, 423, 424, 568
see also INGRES
query (OO), 679
discipline, 680
query decomposition, 514ff
query language, 8
query language processor, 8
query modification
see request modification
query plan, 507
query tree, 504
Query-By-Example
see QBE
QUIST, 533
quota query, 213
quotient relation, 177

R*, 596, 608, 624ff
radix search tree
see trie
Räihä, Kari-Jouke, 369
Ramakrishna, M. V., 751, 753
Ramakrishnan, Raghu, 532, 810, 812
Raman, V. K., 756
RANGE, 188
QUEL, 186

range variable, 186
  see also domain variable, tuple variable
Rashid, Louiqa, 813
RDA, 618
read lock
  see S lock
rear compression
  see compression
REBIND (DB2), 768
recompilation (DB2)
  see compilation (DB2)
record ID
  see RID
record number
  see RID
record prefix, 724
recovery, 374ff
  distributed, 610
  media, 382
  system, 380
  transaction, 377
recursive queries, 795, 798ff, 806
  see also bill of materials, parts explosion
reducing the search space, 507
reductio ad absurdum, 781
redundancy, 14, 274, 288, 324, 326, 374
  controlled, 14
  strong, 101, 321
referenced relation, 118
referencing relation, 118
referential constraint, 118
referential cycle, 119
referential diagram, 118
referential integrity (metarule), 118, 120
  OO, 676
  SQL, 223
referential path, 119
regular entity (E/R), 351
Reiner, David S., 526
Reiser, A., 410
Reisner, Phyllis, 255
Reiter, Raymond, 776, 810
relation, 56, 86ff
  base, 95
  Codd, 102
  derived, 95
  expressible, 95
  meaning, 97, 168
  named, 95
  predicate, 448
  stored, 96
  "time-varying", 89
  value, 86
  variable, 86
  vs. table, 88
relation-valued attribute
  see attribute
relational algebra, 139ff
  implementation, 519ff
  primitive operators, 158, 178
  purpose, 158
  transformation rules, 159, 508ff
relational assignment, 166
  target relation, 488
relational calculus, 185ff
  see also domain calculus, tuple calculus
relational comparisons, 169
relational completeness, 160, 200
  SQL, 258
relational database, 98
relational model, *passim*
Relational Model / Version 1
  Codd, 103
  comments and criticisms, *passim*
Relational Model / Version 2
  see RM/V2
relational system, 52, 56
relationship, 11–12, 349, 350, 353
  OO, 675
  recursive, 357
  see also entity
reliability, 598
Remote Data Access
  see RDA
remote procedure call
  see RPC
remote request (DRDA), 623
remote unit of work (DRDA), 623
RENAME, 143, 144
reorganization, 44
repeatable read
  SQL *vs.* DB2, 403
  see also isolation level
repeating group, 56, 744
REPLACE (QUEL), 424
replication, 602
  vs. two-phase commit, 611
replication independence, 603
replication transparency
  see replication independence
repository
  see dictionary
request modification, 423
Rescher, Nicholas, 591

resolution, 780, 781, 787, 799
Resource Limit Facility (DB2), 771
resource manager, 382
restrict, 52, 67, 141, 150
RESTRICT (SQL)
  see CASCADE
RESTRICTED
  see foreign key rules
restriction condition, 150
restructuring, 468
RETRIEVE (QUEL), 423
reusability (code), 554, 645
Reuter, Andreas, 388
REVOKE (SQL), 433
RID, 715, 721
  DB2, 763
Rissanen, Jorma, 305, 316, 324, 329
Rivest, R. L., 428, 437
RM/T, 349, 367, 565
  comments and criticisms, 105
RM/V2, 103
Robinson, John T., 412, 413
Robson, David, 649
Roddick, John F., 684
ROLLBACK, 376
  SQL, 384
Rosenthal, Arnon, 813
Roth, Mark A., 566
Rothermel, Kurt, 413
Rothnie, James B., Jr., 529, 623, 624, 625
Roussopoulos, Nick, 625, 756
Rowe, Lawrence A., 536, 708
RPC, 620
RR
  see isolation level
RSA encryption
  see public-key encryption
Rubinstein, Brad, 568
rule of inference
  see deductive axiom
Run Time Supervisor (DB2), 764, 767
RUNSTATS (DB2), 514, 770
run-time binding, 556

S lock, 395ff
SAA, 255
Saccà, Domenico, 812
Sacco, Giovanni Maria, 528
safe expression, 211
saga, 387, 672
Sagiv, Yehoshua, 344, 534, 705, 811
Salem, Kenneth, 387
Salveter, Sharon C., 459
Salzberg, Betty, 756
Samet, Hanan, 755, 756
Sandhu, Ravi, 436
Sarin, S. K., 459
savepoint, 388, 672
Sbatella, L., 257
scalar, 55, 547ff
scalar expression (SQL), 243
scatter table, 751
schedule, 401
  equivalent, 401
  interleaved, 401
  serial, 401
Schek, H., 565
Schkolnick, Mario, 755
Schmid, H. A., 370
Schwartz, Peter, 389
Sciore, E., 344
SDD-1, 596, 623ff
second normal form, 300
secondary index
  see index
security, 416ff
  context-dependent, 422
  QUEL, *see* request modification
  SQL, 430ff
  statistical summary, 421
  value-dependent, 420, 421
  value-independent, 420
  via views, 470
security rules, 417ff
SELECT (OPAL), 663
select (relational algebra)
  see restrict
SELECT (SQL), 66, 226ff
"SELECT *" (SQL), 227
SELECT clause (SQL), 235
select-expression (SQL), 235ff
  conceptual evaluation, 239ff
Selinger, Patricia G., 436, 534, 626
Sellis, Timos K., 539, 540, 625
semantic modeling, 347
semantic optimization
  see optimization
semantic override
  see domain check override
semijoin, 529, 626
seminaive evaluation, 802
Senko, Michael E., 749
SEQUEL, 255
SEQUEL/2, 256

SEQUEL/XRM, 255
sequence set (index), 730
sequential access (index), 726
sequential access (physical)
  see physical sequence
serializability, 400ff
serialization, 402
server
  see client/server
SET CONNECTION (SQL), 618
SET CONSTRAINTS (SQL), 455
SET DEFAULT (SQL)
  see CASCADE
set membership
  see IN
SET NULL (SQL)
  see CASCADE
set processing capability, 54
SET TRANSACTION (SQL), 384
SET_*v*, 659
Sevcik, Kenneth C., 755, 756
Severance, Dennis G., 749, 754
shadow page, 389
Shamir, A., 428, 437
Shapiro, Leonard J., 528
shared
  see data sharing
shared lock
  see S lock
Shaw, Gail M., 706
Shenoy, Sreekumar T., 533
Shibamiya, A., 535
Shim, Kyuseok, 540
Shipman, David W., 623, 624, 625, 706
Shneiderman, Ben, 751
Siegel, Michael, 533
Silberschatz, Abraham, 388, 566, 624, 749, 813
Simon, Alan R., 255
simple domain
  see domain
SIMULA 67, 649
single-user system, 5
singleton SELECT (SQL), 247
Skeen, J., 684
Skolem constant, 787
Skolem function, 787
Sloan, Alan, D., 757
Smalltalk-80, 649, 652
Smith, Diane C. P., 770
Smith, John Miles, 338, 344, 370, 530
Smith, Ken, 437
Smith, Peter D., 749
snapshot, 96, 489, 494
soft crash, 380
Sol, H. G., 370
sort/merge join
  see join implementation
Sowa, John F., 809
spatial data, 691, 755–757
sphere of control, 387
Spyratos, Nicolas, 813
SQL, *passim*
SQL standard
  see SQL/92
SQL2
  see SQL/92
SQL-92
  see SQL/92
SQL/92, 219
SQL/DS, 256
SQLCODE, 246
SQLSTATE, 246
SQUARE, 255
Starburst, 539, 540, 708
static filtering, 803
statistical database, 435
Stein, Jacob, 650
Stoll, Robert R., 809
Stonebraker, Michael R., 210, 437, 459, 496, 536, 567, 568, 624, 627, 650, 706, 708, 755, 810
storage structure, 712ff
  DB2, 762
Stored Data Manager (DB2), 764, 767
stored field, 18
stored file, 19
stored procedure, 122, 441, 620
stored record, 19
  DB2, 763
stored record management, 720ff
stored relation
  see relation
stored table (DB2), 763
Storey, Veda C., 370
strict homogeneity assumption, 595
Strnad, Alois J., 176
Strong, H. Raymond, 752
strong redundancy
  see redundancy
strong typing, 546
Stroustrup, Bjarne, 684
Structured Query Language
  see SQL

Su, Stanley Y. W., 813
subclass
  see subtype
subquery (SQL), 230, 231, 536ff
substitution (view processing), 467
substitutability (subtype), 554, 645
subtype, 350, 354, 368, 554ff
SUM
  see aggregate function
summarize, 163
  see also generalized summarize
Summers, Rita C., 435, 436
Sunderraman, Rajshakhar, 591
Sundgren, Bo, 370
superclass
  see subtype
supertype
  see subtype
surrogate, 368
Swami, Arun, 539
Swenson, J. R., 370
swizzling, 674
SYBASE, 133, 568
symmetric exploitation, 673
syncpoint
  see commit point
system catalog
  see catalog
system failure, 350
System R, 210, 256, 388, 389, 436, 494, 528, 534, 537, 773

table, 56
  vs. relation, *see* relation
table expression (SQL), 226, 235ff
table space (DB2), 762
TABLE_DEE and TABLE_DUM, 109, 138, 179, 180, 214, 589
tableau, 533
Tanca, L., 814
target item, 192
target relation
  assignment, *see* relational assignment
  referential integrity, *see* referenced relation
Tasker, Dan, 370
Taylor, Elizabeth, 363
Teorey, Toby J., 354, 371, 748
Tezuka, M., 531
theorem, 776, 791
θ-join, 154
θ-restrict
  see restrict
Third Generation Database System Manifesto, 686, 706
third normal form, 296, 302, 312
Thomasian, Alexander, 412
three-valued logic, 123, 570ff
  interpretation, 579
Thuraisingham, Bhavani, 437
Tiberio, P., 755
TIMES
  see Cartesian product
timestamping, 411
Todd, Stephen J. P., 175, 177, 368
Traiger, Irving L., 412, 413, 495
transaction, 375
  unit of concurrency, 379
  unit of integrity, 379
  unit of recovery, 379
  unit of work, 375
transaction manager, 376
transformation rules
  see relational algebra
transitive closure, 802
transitive FD
  see FD
tree-structured index
  see index
trie, 749, 754
trie hashing
  see hashing
trigger condition, 453
triggered procedure, 122, 441, 453
trivial FD
  see FD
trusted system, 426
Tsichritzis, Dionysios C., 50, 650
Tsur, Shalom, 812, 814
tuple, 57, 79, 87
tuple calculus, 186, 187ff
tuple ID, 369
tuple substitution
  see query decomposition
tuple variable, 186, 188
Turbyfill, Carolyn, 528
two-phase commit, 382, 604, 611
  improvements, 387
two-phase locking protocol, 401
two-phase locking theorem, 401
type compatible, 145, 543, 546, 552, 556
type constructor, 551
  tuple, 553
  relation, 551

type hierarchy, 354, 556, 645ff
type lattice, 559, 647

Uhrowczik, Peter P., 51
Ullman, Jeffrey D., 211, 306, 330, 339, 342, 343, 344, 749, 809, 811, 812
unary relation
  see degree
uncommitted dependency, 392, 397
unification, 788, 799
union, 141, 146
  SQL, 234
union compatible, 145
  see also type compatible
UniSQL, 694ff, 707
universal quantifier
  see FORALL
universal relation, 315, 342, 343
universe of discourse, 784
unk
  see unknown
UNK
  in domain, 575
  vs. unk, 575
  see also null, three-valued logic
unknown (truth value), 570
  see also three-valued logic
unload/reload, 44
  see also dump/restore
UNNEST, 561, 567
unplanned request, 40
UPDATE, 167
  CURRENT (SQL), 250
  embedded (SQL), 248
  SQL, 67, 234
update anomalies, 298, 301, 308, 327, 332
update propagation, 15
  distributed system, 603, 610
update rule
  see foreign key rules
updating views
  see view
user group, 418
user ID, 418
utilities, 44
  DB2, 769

Valduriez, Patrick, 625, 798, 810
value set (E/R), 353
Van Gelder, Allen, 813
van Griethuysen, J. J., 50
Vardi, Moshé Y., 342, 343
Verrijn-Stuart, A. A., 370
version, 671
view, 62, 96, 466ff
  OO, 681
  retrieval, 472
  security (SQL), 430
  update, 472ff
  vs. base relation, 64
violation response
  integrity, 443
  security, 419
virtual hashing
  see hashing
Virtual Storage Access Method
  see VSAM
von Bültzingsloewen, Günter, 536, 537
von Hallé, Barbara, 368
Vossen, Gottfried, 340, 650
VSAM, 730, 750

Wade, Bradford, W., 436, 773
Wagner, R. E., 750
Wait-For Graph, 400, 614
Waldinger, Richard, 809
Walker, Adrian, 349, 809
Warden, Andrew
  see Darwen, Hugh A. C.
Warren, David H. D., 538
weak entity (E/R), 351
well-formed formula
  see WFF
WFF, 188, 783
Whang, Kyu-Young, 540, 813
WHENEVER (SQL), 246
WHERE clause (SQL), 238
White, Colin J., 132, 773
Widom, Jennifer, 458
Wiederhold, Gio, 748
Wilkinson, W. Kevin, 529
Williams, Robin, 627
Wilson, Walter G., 809
Winsberg, Paul, 51, 366
Winslett, Marianne, 437
Wisconsin benchmark, 528
Wodon, P., 210
Wolfson, Ouri, 813
Wong, Eugene, 210, 437, 459, 535, 538, 624, 627, 754
Wong, Harry K. T., 536, 537, 650
Wood, C., 435, 436
WORM disk
  see optical disk

Worthington, P. S., 535
write lock
  see X lock
write-ahead log rule, 379

X lock, 395ff
X/Open, 255

Yang, Dongqing, 354, 371
Yannakakis, M., 534
Yao, S. Bing, 527, 750
Yost, Robert A., 256
Youn, Cheong, 813
Youssefi, Karel, 535
Yu, C. T., 627, 813
Yuen, P. S. T., 751

Zaniolo, Carlo, 312, 316, 707, 812
Zdonik, Stanley B., 650, 706
Zhang, Weining, 813
Zloof, Moshé M., 210–211

S

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

P

P#	PNAME	COLOR	WEIGHT	CITY
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

SP

S#	P#	QTY
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200
S4	P4	300
S4	P5	400

The suppliers-and-parts database (sample values)

S

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

P

P#	PNAME	COLOR	WEIGHT	CITY
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

J

J#	JNAME	CITY
J1	Sorter	Paris
J2	Display	Rome
J3	OCR	Athens
J4	Console	Athens
J5	RAID	London
J6	EDS	Oslo
J7	Tape	London

SPJ

S#	P#	J#	QTY
S1	P1	J1	200
S1	P1	J4	700
S2	P3	J1	400
S2	P3	J2	200
S2	P3	J3	200
S2	P3	J4	500
S2	P3	J5	600
S2	P3	J6	400
S2	P3	J7	800
S2	P5	J2	100
S3	P3	J1	200
S3	P4	J2	500
S4	P6	J3	300
S4	P6	J7	300
S5	P2	J2	200
S5	P2	J4	100
S5	P5	J5	500
S5	P5	J7	100
S5	P6	J2	200
S5	P1	J4	100
S5	P3	J4	200
S5	P4	J4	800
S5	P5	J4	400
S5	P6	J4	500

The suppliers-parts-projects database (sample values)