

FUNDAMENTALS OF DATABASE SYSTEMS

— ELMASRI / NAVATHE —

SECOND EDITION



FUNDAMENTALS OF DATABASE SYSTEMS

FUNDAMENTALS OF DATABASE SYSTEMS

SECOND EDITION

Ramez Elmasri

Department of Computer Science Engineering
University of Texas at Arlington

Shamkant B. Navathe

College of Computing
Georgia Institute of Technology



The Benjamin/Cummings Publishing Company, Inc.
Redwood City, California • Menlo Park, California
Reading, Massachusetts • New York • Don Mills, Ontario
Wokingham, U.K. • Amsterdam • Bonn • Sydney
Singapore • Tokyo • Madrid • San Juan

Executive Editor: Dan Joraanstad
Editorial Assistant: Laura Cheu
Production Editor: George Calmenson, The Book Company
Production Coordinator: Teri Holden
Marketing Manager: Mary Tudor
Manufacturing Coordinator: Janet Weaver
Text Design: Hal Lockwood
Cover Design: Yvo Riezebos
Copy Editor: Steven Gray
Proofreader: Martha Ghent
Composition and Illustrations: GTS Graphics, Inc.

Cover Art: "Bloom"—Original hand-pulled limited edition serigraph by Tetsuro Sawada. Exclusive Sawada publisher and distributor: Buschlen/Mowatt Fine Arts Ltd., Main Floor, 1445 West Georgia Street, Vancouver, Canada V6G 2T3 (604) 682-1234.

Copyright © 1994 by Ramez Elmasri and Shamkant B. Navathe

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

Library of Congress Cataloging-in-Publication Data

Elmasri, Ramez.

Fundamentals of database systems / Ramez Elmasri, Shamkant B.

Navathe.

p. cm.

Includes bibliographical references and index.

ISBN 0-8053-1748-1

1. Data base management. 2. Data base design. I. Navathe, Sham.

II. Title.

QA76.9.D3E57 1994

005.74—dc20

93-36697

ISBN 0-8053-1748-1

1 2 3 4 5 6 7 8 9 10-DO—98 97 96 95 94

The Benjamin/Cummings Publishing Company, Inc.
390 Bridge Parkway
Redwood City, CA 94065

Record Storage and Primary File Organizations

In Chapters 4 and 5, we turn our attention to the techniques used for physical storage of data on the computer system. Databases are typically organized into files of records, which are stored on computer disks. We start in Section 4.1 by introducing the concepts of computer storage hierarchies. In Section 4.2 we describe disk storage devices and their characteristics, and we also briefly describe tape storage devices. Section 4.3 covers the technique of double buffering, which is used to speed up retrieval of multiple disk blocks. In Section 4.4 we discuss various ways of formatting and storing records of a file on disk. In Section 4.5 we present the various types of operations that are typically applied to records of a file. We then present three primary methods for organizing records of a file on disk: unordered records, discussed in Section 4.6; ordered records, in Section 4.7; and hashed records, in Section 4.8.

Section 4.9 very briefly discusses files of mixed records and other primary methods for organizing records, such as B-trees. In Chapter 5 we discuss techniques for creating access structures, called indexes, that speed up searching for and retrieving records. These techniques involve storage of auxiliary data besides the records themselves.

Chapters 4 and 5 may be browsed through or even skipped entirely by readers who have already studied file organizations. They can also be postponed and read later. The material covered here is necessary for understanding some of the later chapters in the book—in particular, Chapters 14 and 16 through 19.

4.1 Introduction

The collection of data that makes up a computerized database must be physically stored on some computer **storage medium**. The DBMS software can then retrieve, update, and process this data as needed. Computer storage media form a *storage hierarchy* that includes two main categories:

- Primary storage. This category includes storage media that can be operated on directly by the computer central processing unit (CPU), such as the computer main memory and smaller but faster cache memories. Primary storage usually provides fast access to data but is of limited storage capacity.
- Secondary storage. Secondary storage devices include magnetic disks, optical disks, tapes, and drums and usually are of larger capacity, cost less, and provide slower access to data than do primary storage devices. Data in secondary storage cannot be processed directly by the CPU; it must first be copied into primary storage.

Databases typically store large amounts of data that must persist over long periods of time. The data is accessed and processed repeatedly during this period. This contrasts with the notion of data structures that persist for only a limited time during program execution, which is common in programming languages. Most databases are stored permanently on **magnetic disk** secondary storage, for the following reasons:

- Generally, databases are too large to fit entirely in main memory.
- The circumstances that cause permanent loss of stored data arise less frequently for disk secondary storage than for primary storage. Hence, we refer to disk—and other secondary storage devices—as **nonvolatile storage**, whereas main memory is often called **volatile storage**.
- The cost of storage per unit of data is an order of magnitude less for disk than for primary storage.

New technologies are emerging, including optical disk storage, cheaper and larger main memories, and special-purpose database-oriented hardware. These technologies may provide viable alternatives to the use of magnetic disks in the future. For now, however, it is important to study and understand the properties and characteristics of magnetic disks and the way data files can be organized on disk in order to design effective databases with acceptable performance.

Magnetic tapes are frequently used as a storage medium for backing up the database because storage on tape costs even less than storage on disk. However, access to data on tape is quite slow. Data stored on tapes is **off-line**; that is, some intervention by an operator (or a loading device) to load a tape is needed before this data becomes available. In contrast, disks are **on-line** devices that can be directly accessed at any time.

In this and the following chapter, we describe the techniques used to store large amounts of structured data on disk. These techniques are important for database designers, the DBA, and implementers of a DBMS. Database designers and the DBA must know the advantages and disadvantages of each storage technique when they design, implement, and operate a database on a specific DBMS. Usually, the DBMS will have several options available for organizing the data, and the process of **physical database design**

involves choosing the particular data organization techniques that best suit the given application requirements from among the available options. DBMS system implementers must study data organization techniques so that they can implement them efficiently and thus provide the DBA and users of the DBMS with sufficient options.

Typical database applications need only a small portion of the database at a time for processing. Whenever a certain portion of the data is needed, it must be located on disk, copied to main memory for processing, and then rewritten to the disk if the data is changed. The data stored on disk is organized as **files** of **records**. Each record is a collection of data values that can be interpreted as facts about entities, their attributes, and their relationships. Records should be stored on disk in a manner that makes it possible to locate them efficiently whenever they are needed.

There are several **primary file organizations** which determine how the records of a file are *physically placed on the disk*. A *heap file* (or *unordered file*) places the records on disk in no particular order, whereas a *sorted file* (or *sequential file*) keeps the records ordered by the value of a particular field. A *hashed file* uses a hash function to determine record placement on disk. Other primary file organizations, such as *B-trees*, use tree structures. We discuss primary file organizations in Sections 4.6 to 4.9.

4.2 Secondary Storage Devices

In this section we describe some characteristics of magnetic disk and magnetic tape storage devices. Readers who have studied these devices before may just browse through this section.

4.2.1 Hardware Description of Disk Devices

Magnetic disks are used for storing large amounts of data. The most basic unit of data on the disk is a single **bit** of information. By magnetizing an area on disk in certain ways, one can make it represent a bit value of either 0 (zero) or 1 (one). To code information, bits are grouped into **bytes** (or **characters**). Byte sizes are typically 4 to 8 bits, depending on the computer and the device. We assume that one character is stored in a single byte, and we use the terms *byte* and *character* interchangeably. The **capacity** of a disk is the number of bytes it can store, which is usually very large. We refer to disk capacities in kilobytes (Kbyte or 1000 bytes), megabytes (Mbyte or 1 million bytes), and gigabytes (Gbyte or 1 billion bytes). Small floppy disks used with microcomputers typically hold from 400 Kbytes to 1.2 Mbytes; hard disks for micros typically hold from 30 to 250 Mbytes; and large disk packs used with minicomputers and mainframes have capacities that range up to a few Gbytes. Disk capacities continue to grow as technology improves.

Whatever their capacity, disks are all made of magnetic material shaped as a thin circular disk (Figure 4.1(a)) and protected by a plastic or acrylic cover. A disk is **single-sided** if it stores information on only one of its surfaces and **double-sided** if both surfaces are used. To increase storage capacity, disks are assembled into a **disk pack** (Figure 4.1(b)), which may include as many as 30 surfaces. Information is stored on a disk surface in concentric circles of *small width*, each having a distinct diameter. Each circle is called a **track**. For disk packs, the tracks with the same diameter on the various surfaces are

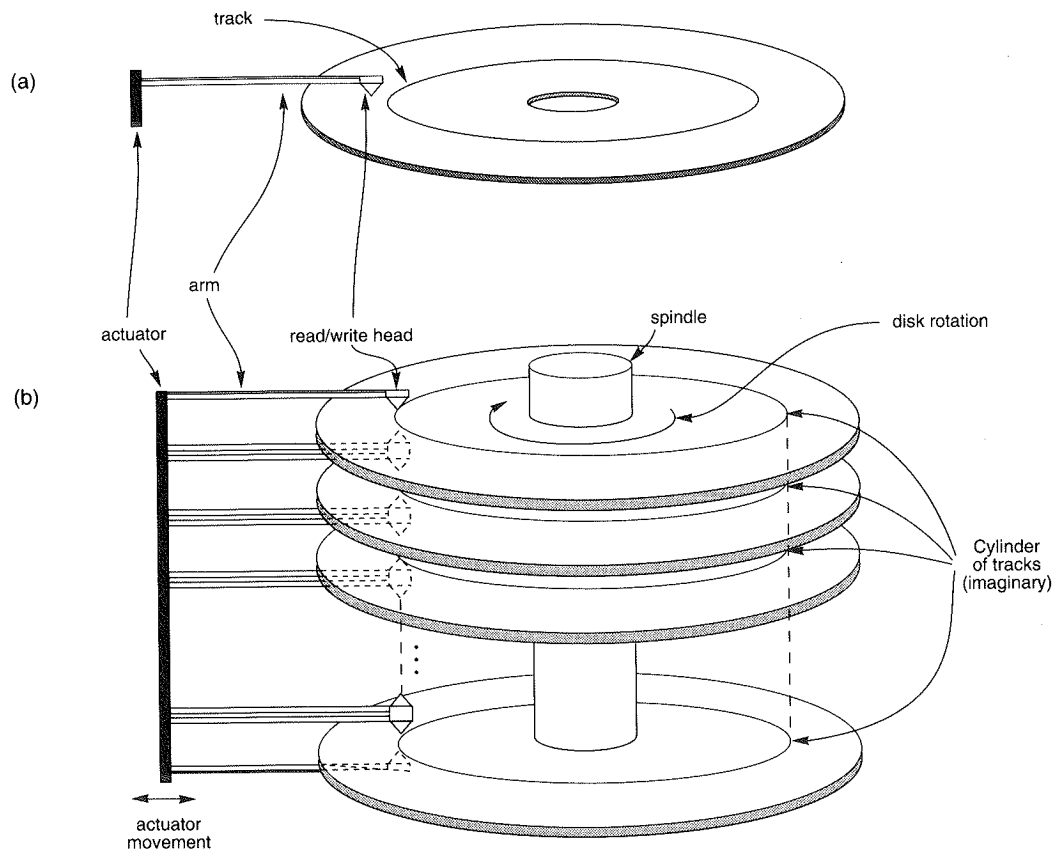


Figure 4.1 (a) A single-sided disk with read/write hardware. (b) A disk pack with read/write hardware.

called a **cylinder** because of the shape they would form if connected in space. The concept of a cylinder is important because data stored on the same cylinder can be retrieved much faster than if it were distributed among different cylinders.

Each concentric circle typically stores the same amount of information, so bits are packed more densely on the smaller-diameter tracks. The number of tracks on a disk ranges up to 800, and the capacity of each track typically ranges from 4 to 50 Kbytes. Because a track usually contains a large amount of information, it is divided into smaller blocks or sectors. The division of a track into **sectors** is hard-coded on the disk surface and cannot be changed. Sectors subtend a fixed angle at the center (Figure 4.2), and not all disks have their tracks divided into sectors. The division of a track into equal-sized **blocks** or *pages* is set by the operating system during disk **formatting** (or **initialization**). Block size is fixed during initialization and cannot be changed dynamically. Typical disk block sizes range from 512 to 4096 bytes. A disk with hard-coded sectors often has the sectors further subdivided into blocks during initialization. Blocks are separated

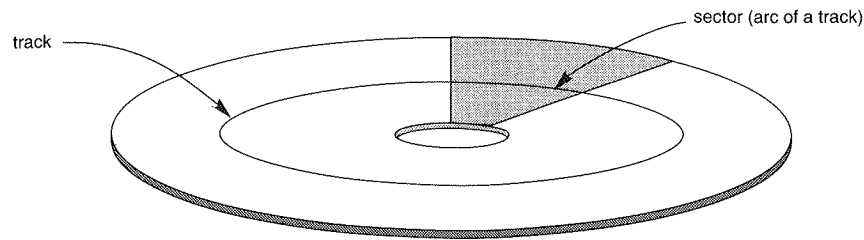


Figure 4.2 A group of sectors that subtend the same angle.

by fixed-size **interblock gaps**, which include specially coded control information written during disk initialization. This is used to determine which block on the track follows each interblock gap.

A disk is called a *random access* addressable device. Transfer of data between main memory and disk takes place in units of blocks. The **hardware address** of a block—a combination of a surface number, track number (within the surface), and block number (within the track)—is supplied to the disk input/output (I/O) hardware. The address of a **buffer**—a contiguous reserved area in main storage that holds one block—is also provided. For a **read** command, the block from disk is copied into the buffer; whereas for a **write** command, the contents of the buffer are copied into the disk block. Sometimes several contiguous blocks, called a **cluster**, may be transferred as a unit. In this case the buffer size is adjusted to match the number of bytes in the cluster.

The actual hardware mechanism that reads or writes a block is the disk **read/write head**, which is part of a system called a **disk drive**. A disk or disk pack is mounted in the disk drive, which includes a motor that rotates the disks. A read/write head includes an electronic component attached to a **mechanical arm**. Disk packs with multiple surfaces are controlled by several read/write heads—one for each surface (see Figure 4.1(b)). All arms are connected to an **actuator** attached to another electrical motor, which moves the read/write heads in unison and positions them precisely over the cylinder of tracks specified in a block address.

Disk drives for hard disks rotate the disk pack continuously at a constant speed. For a floppy disk, the disk drive begins to rotate the disk whenever a particular read or write request is initiated and ceases rotation soon after the data transfer is completed. Once the read/write head is positioned on the right track and the block specified in the block address moves under the read/write head, the electronic component of the read/write head is activated to transfer the data. Some disk units have fixed read/write heads, with as many heads as there are tracks. These are called **fixed-head** disks, whereas disk units with an actuator are called **movable-head** disks. For fixed-head disks, a track or cylinder is selected by electronically switching to the appropriate read/write head rather than by actual mechanical movement; consequently, it is much faster. However, the cost of the additional read/write heads is quite high, so fixed-head disks are not commonly used.

To transfer a disk block, given its address, the disk drive must first mechanically position the read/write head on the correct track. The time required to do this is called the **seek time**. Following that, there is another delay—called the **rotational delay** or

latency—while the beginning of the desired block rotates into position under the read/write head. Finally, some additional time is needed to transfer the data; this is called the **block transfer time**. Hence, the total time needed to locate and transfer an arbitrary block, given its address, is the sum of the seek time, rotational delay, and block transfer time. The seek time and rotational delay are usually much larger than the block transfer time. To make the transfer of multiple blocks more efficient, it is common to transfer several consecutive blocks on the same track or cylinder. This eliminates the seek time and rotational delay for all but the first block and can result in a substantial saving of time when numerous contiguous blocks are transferred. Usually, the disk manufacturer provides a **bulk transfer rate** for calculating the time required to transfer consecutive blocks. Appendix B contains a discussion of these and other disk parameters.

The time needed to locate and transfer a disk block is in the order of milliseconds, usually ranging from 15 to 60 msec. For contiguous blocks, locating the first block takes from 15 to 60 msec, but transferring subsequent blocks may only take 1 to 2 msec each. Many search techniques take advantage of consecutive retrieval of blocks when searching for data on disk. In any case, a transfer time in the order of milliseconds is considered quite high compared to the time required to process data in main memory by current CPUs. Hence, locating data on disk is a *major bottleneck* in database applications. The file structures we discuss here and in Chapter 5 attempt to *minimize the number of block transfers* needed to locate and transfer the required data from disk to main memory.

4.2.2 Magnetic Tape Storage Devices

Disks are **random access** secondary storage devices, because an arbitrary disk block may be accessed “at random” once we specify its address. Magnetic tapes are **sequential access** devices; to access the n^{th} block on tape, we must first read the preceding $n - 1$ blocks. Data is stored on reels of high-capacity magnetic tape, somewhat similar to audio or video tapes. A **tape drive** is required to read the data from or to write the data to a **tape reel**. Usually, each group of bits that form a byte is stored across the tape, and the bytes themselves are stored consecutively on the tape.

A read/write head is used to read or write data on tape. Data records on tape are also stored in blocks—although the blocks may be substantially larger than those for disks, and interblock gaps are also quite large. With typical tape densities of 1600 to 6250 bytes per inch, a typical interblock gap* of 0.6 inches corresponds to 960 to 3750 bytes of wasted storage space. For better space utilization it is customary to group many records together in one block.

The main characteristic of a tape is its requirement that we access the data blocks in **sequential order**. To get to a block in the middle of a reel of tape, we must mount the tape and then read through it until the required block gets under the read/write head. For this reason, tape access can be slow and tapes are not used to store on-line data, except for some specialized applications. However, tapes serve a very important function—that of **backing up** the database. One reason for backup is to keep copies of disk files in case the data is lost because of a disk crash, which can happen if the disk read/write head touches the disk surface because of mechanical malfunction. For this reason,

*Called *interrecord gaps* in tape terminology.

disk files are periodically copied to tape. Tapes can also be used to store excessively large database files. Finally, database files that are seldom used or outdated but are required for historical record keeping can be **archived** on tape. Recently, smaller 8-mm magnetic tapes (similar to those used in camcorders) and CD-ROMs (compact disks—read only memory) have become popular media for backing up data files from workstations and personal computers and for storing images and system libraries.

4.3 Buffering of Blocks

When several blocks need to be transferred from disk to main memory and all the block addresses are known beforehand, several buffers can be reserved in main memory to speed up the transfer. While one buffer is being read or written, the CPU can process data in the other buffer. This is possible because an independent disk input/output (I/O) processor usually exists that, once started, can proceed to transfer a data block between memory and disk independent of and in parallel to CPU processing.

Figure 4.3 illustrates how two processes can proceed in parallel. Processes A and B are running **concurrently** in an **interleaved** fashion, whereas processes C and D are running **concurrently** in a **simultaneous** fashion. When a single CPU controls multiple processes, simultaneous execution is not possible. However, the processes can still run concurrently in an interleaved way. Buffering is most useful when processes can run concurrently in a simultaneous fashion, either because a separate disk I/O processor is available or because multiple processors exist.

Figure 4.4 illustrates how reading and processing can proceed in parallel when the time required to process a disk block in memory is less than the time required to read the next block and fill a buffer. The CPU can start processing a block once its transfer

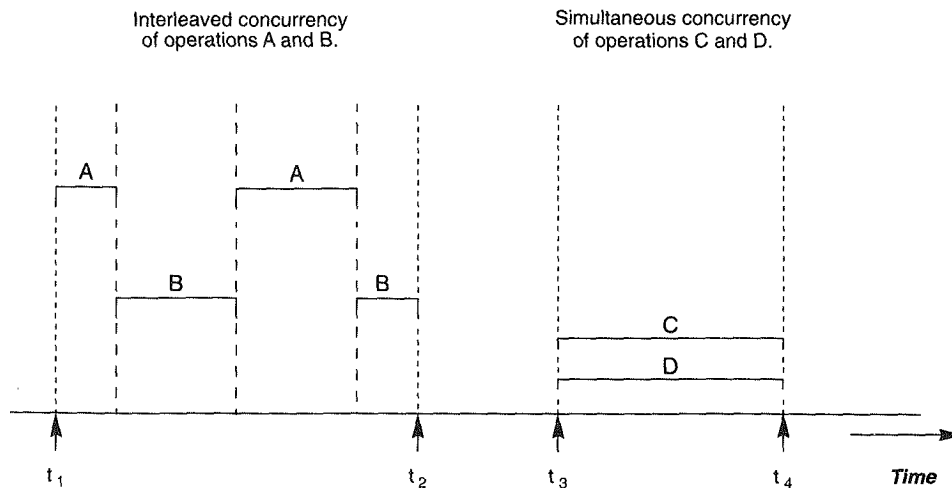


Figure 4.3 Interleaved versus simultaneous concurrency.

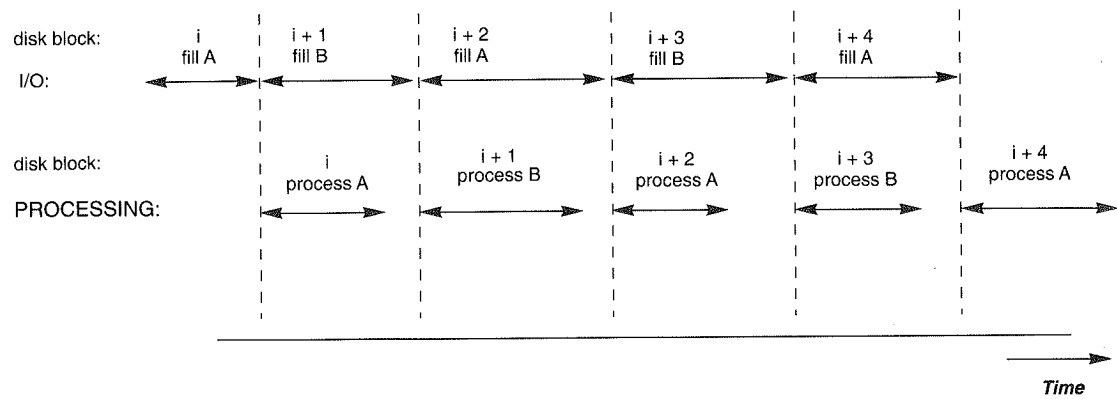


Figure 4.4 Use of two buffers, A and B, for reading from disk.

to main memory is completed; at the same time the disk I/O processor can be reading and transferring the next block into a different buffer. This technique is called **double buffering** and can also be used to write a continuous stream of blocks from memory to the disk. Double buffering permits continuous reading or writing of data on consecutive disk blocks, which eliminates the seek time and rotational delay for all but the first block transfer. Moreover, data is kept ready for processing, thus reducing the waiting time in the programs.

4.4 Placing File Records on Disk

In this section we define the concepts of records, record types, and files. Then we discuss different techniques for placing file records on disk.

4.4.1 Record Types

Data is usually stored in the form of **records**. Each record consists of a collection of related data **values** or **items**, where each value is formed of one or more bytes and corresponds to a particular **field** of the record. Records usually describe entities and their attributes. For example, an **EMPLOYEE** record represents an employee entity, and each field value in the record specifies some attribute of that employee, such as **NAME**, **BIRTH-DATE**, **SALARY**, or **SUPERVISOR**. A collection of field names and their corresponding data types constitutes a **record type** or **record format** definition. A **data type**, associated with each field, specifies the type of values a field can take.

The data type of a field is usually one of the standard data types used in programming. These include numeric (integer, long-integer, or real-number), string of characters (fixed-length or varying), Boolean (having 0 and 1 or **TRUE** and **FALSE** values only), and sometimes specially coded **date** and **time** data types. The number of bytes required for each data type is fixed for a given computer system. An integer may require 4 bytes, a long integer 8 bytes, a real number 4 bytes, a Boolean 1 byte, a date 4 bytes (to code

the date into an integer), and a fixed-length string of k characters k bytes. Variable-length strings may require as many bytes as there are characters in each field value. For example, an EMPLOYEE record type may be defined—using PASCAL notation—as follows:

RECORD TYPE NAME	FIELD NAMES	DATA TYPES
type EMPLOYEE = record	NAME	: packed array [1..30] of character;
	SSN	: packed array [1..9] of character;
	SALARY	: integer;
	JOB CODE	: integer;
	DEPARTMENT	: packed array [1..20] of character
	end;	

In recent database applications, the need may arise for storing data items that consist of large unstructured objects, which represent images, digitized video or audio streams, or free text. These are referred to as **BLOBs** (binary large objects). Normally, a BLOB data item is stored separately from its record in a pool of disk blocks, and a pointer to the BLOB is included in the record.

4.4.2 Files, Fixed-length Records, and Variable-length Records

A **file** is a *sequence* of records. In many cases, all records in a file are of the same record type. If every record in the file has exactly the same size (in bytes), the file is said to be made up of **fixed-length** records. If different records in the file have different sizes, the file is said to be made up of **variable-length** records. A file may have variable-length records for several reasons:

- The file records are of the same record type, but one or more of the fields are of varying size (**variable-length fields**). For example, the NAME field of EMPLOYEE can be a variable-length field.
- The file records are of the same record type, but one or more of the fields may have multiple values for individual records; such a field is called a **repeating field** and a group of values for the field is often called a **repeating group**.
- The file records are of the same record type, but one or more of the fields are **optional**; that is, they may have values for some but not all of the file records (**optional fields**).
- The file contains records of *different record types* and hence of varying size (**mixed file**). This would occur if related records of different types were placed together on disk blocks; for example, the GRADE_REPORT records of a particular student may be placed following that STUDENT's record.

The fixed-length EMPLOYEE records in Figure 4.5(a) have a record size of 71 bytes. Every record has the same fields, and field lengths are fixed, so we can identify the starting byte position of each field relative to the starting position of the record. This facilitates locating field values by programs that access such files. Notice that it is possible to represent a file that logically should have variable-length records as a fixed-length records file. For example, in the case of optional fields we could have *every field* included in *every file record* but store a special null value if no value exists for that field. For a

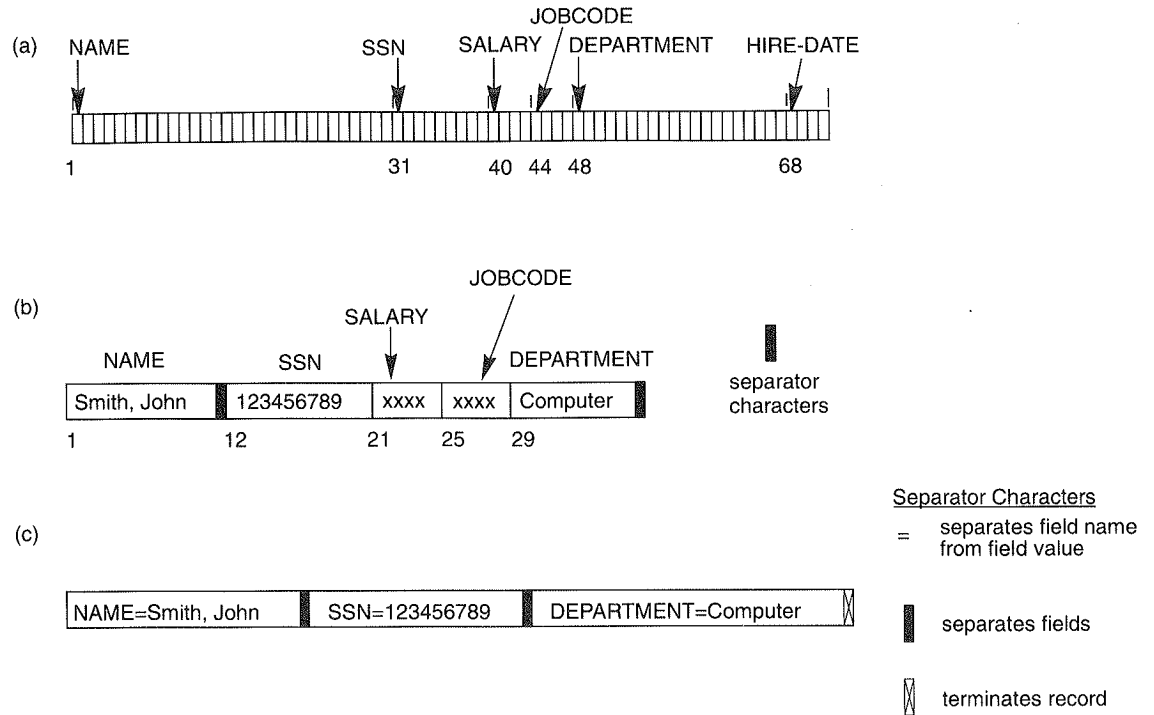


Figure 4.5 Different record storage formats. (a) A fixed-length record with six fields and size of 71 bytes. (b) A record with two variable-length fields and three fixed-length fields. (c) A variable-field record with three types of separator characters.

repeating field, we could allocate as many spaces in each record as the maximum number of values that the field can take. In either case, space is wasted when certain records do not have values for all the physical spaces provided in each record. We now consider other options for formatting records of a file of variable-length records.

For *variable-length fields*, each record has a value for each field, but we do not know the exact length of some field values. To determine the bytes within a particular record that represent each field, we can use special **separator** characters—which do not appear in any field value (such as ? or % or \$)—to terminate variable-length fields (Figure 4.5(b)), or we can store the length of the field in the record.

A file of records with *optional fields* can be formatted in different ways. If the total number of fields for the record type is large but the number of fields that actually appear in a typical record is small, we can include in each record a sequence of <field-name, field-value> pairs rather than just the field values. Three types of separator characters are used in Figure 4.5(c), although we could use the same separator character for the first two purposes—separating the field name from the field value and separating one field from the next field. A more practical option is to assign a short **field type**—say, an integer number—to each field and include in each record a sequence of <field-type, field-value> pairs rather than <field-name, field-value> pairs.

A *repeating field* needs one separator character to separate the repeating values of the field and another separator character to indicate termination of the field. Finally, for a file that includes *records of different types*, each record is preceded by a **record type** indicator. Understandably, programs that process files of variable-length records need to be more complex than those for fixed-length records, where the starting position and size of each field are known and fixed.

4.4.3 Record Blocking and Spanned versus Unspanned Records

The records of a file must be allocated to disk blocks because a block is the *unit of data transfer* between disk and memory. When the block size is larger than the record size, each block will contain numerous records. Some files may have unusually large record sizes that cannot fit in one block. Suppose that the block size is B bytes. For a file of fixed-length records of size R bytes, with $B \geq R$, we can fit $bfr = \lfloor B/R \rfloor$ records per block, where the $\lfloor x \rfloor$ (*floor function*) rounds the value x down to the next integer. The value bfr is called the **blocking factor** for the file. In general, R may not divide B exactly, so we have some unused space in each block equal to

$$B - (bfr * R) \text{ bytes}$$

To utilize this unused space, we can store part of a record on one block and the rest on another block. A **pointer** at the end of the first block points to the block containing the remainder of the record in case it is not the next consecutive block on disk. This organization is called **spanned**, because records can span more than one block. Whenever a record is larger than a block, we *must* use a spanned organization. If records are not allowed to cross block boundaries, the organization is called **unspanned**. This is used with fixed-length records having $B \geq R$ because it makes each record start at a known location in the block, simplifying record processing. For variable-length records, either a spanned or an unspanned organization can be used. If the average record size is large, it is advantageous to use spanning to reduce the lost space in each block. Figure 4.6 illustrates spanned versus unspanned organization.

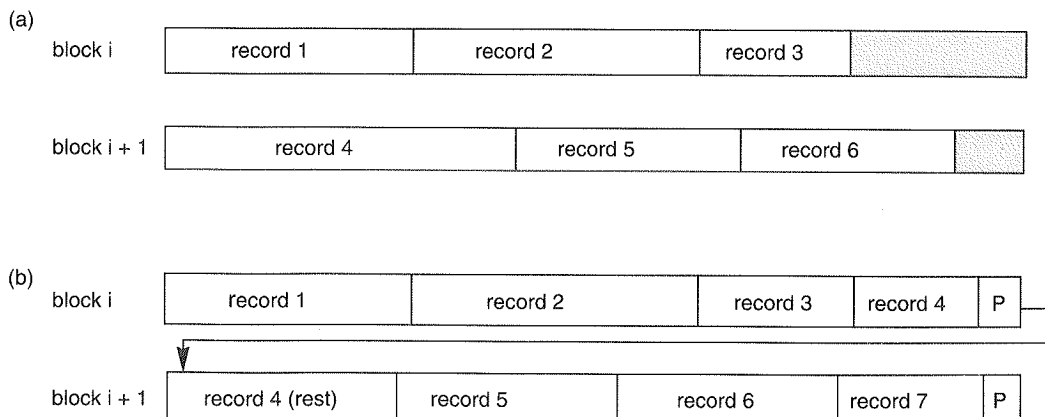


Figure 4.6 Types of record organization. (a) Unspanned. (b) Spanned.

For variable-length records using spanned organization, each block may store a different number of records. In this case, the blocking factor *bfr* represents the *average* number of records per block for the file. We can use *bfr* to calculate the number of blocks *b* needed for a file of *r* records:

$$b = \lceil (r/bfr) \rceil \text{ blocks}$$

where the $\lceil (x) \rceil$ (*ceiling function*) rounds the value *x* up to the next integer.

4.4.4 Allocating File Blocks on Disk

There are several standard techniques for allocating the blocks of a file on disk. In **contiguous allocation** the file blocks are allocated to consecutive disk blocks. This makes reading the whole file very fast using double buffering, but it makes expanding the file difficult. In **linked allocation** each file block contains a pointer to the next file block. This makes it easy to expand the file but makes it slow to read the whole file. A combination of the two allocates **clusters** of consecutive disk blocks, and the clusters are linked together. Clusters are sometimes called **segments** or **extents**. Another possibility is to use **indexed allocation**, where one or more **index blocks** contain pointers to the actual file blocks. It is also common to use combinations of these techniques.

4.4.5 File Headers

A **file header** or **file descriptor** contains information about a file that is needed by the programs that access the file records. The header includes information to determine the disk addresses of the file blocks as well as to record format descriptions, which may include field lengths and order of fields within a record for fixed-length unspanned records and field type codes, separator characters, and record type codes for variable-length records.

To search for a record on disk, one or more blocks are copied into main memory buffers. Programs then search for the desired record or records within the buffers, utilizing the information in the file header. If the address of the block that contains the desired record is not known, the search programs must do a **linear search** through the file blocks. Each file block is copied into a buffer and searched until either the record is located or all the file blocks have been searched unsuccessfully. This can be very time-consuming for a large file. The goal of a good file organization is to locate the block that contains a desired record with a minimal number of block transfers.

4.5 Operations on Files

Operations on files are usually grouped into **retrieval** operations and **update** operations. The former do not change any data in the file, but only locate certain records so that their field values can be examined and processed. The latter change the file by insertion or deletion of records or by modification of field values. In either case, we may have to **select** one or more records for retrieval, deletion, or modification based on a **selection condition**, which specifies criteria that the desired record or records must satisfy.

Consider an EMPLOYEE file with fields NAME, SSN, SALARY, JOBCODE, and DEPARTMENT. A **simple selection condition** may involve an equality comparison on some field value—for example, (SSN = '123456789') or (DEPARTMENT = 'Research'). More complex conditions can involve other types of comparison operators, such as > or ≥; an example is (SALARY ≥ 30000). The general case is to have an arbitrary Boolean expression on the fields of the file as the selection condition.

Search operations on file systems are generally based on simple selection conditions. A complex condition must be decomposed by the DBMS (or the programmer) to extract a simple condition that can be used to locate the records on disk. Each located record is then checked to determine whether it satisfies the full selection condition. For example, we may extract the simple condition (DEPARTMENT = 'Research') from the complex condition ((SALARY ≥ 30000) AND (DEPARTMENT = 'Research')); each record satisfying (DEPARTMENT = 'Research') is located and then tested to see if it also satisfies (SALARY ≥ 30000).

When several file records satisfy a search condition, only the *first* record—with respect to the physical sequence of file records—is located. To locate other records satisfying the condition, additional operations are needed. The most recently located record of the file is designated the **current record**. Subsequent search operations commence from this record and locate the *next* record in the file satisfying the condition.

Actual operations for locating and accessing file records vary from system to system. Below, we present a set of representative operations. Typically, high-level programs, such as DBMS software programs, access the records by using these commands, so we sometimes refer to **program variables** in the following description:

- *Find* (or *Locate*): Searches for the first record satisfying a search condition. Transfers the block containing that record into a main memory buffer (if it is not already there). The record is located in the buffer and becomes the current record. Sometimes, different verbs are used to indicate whether the located record is to be retrieved (*Find*) or updated (*Locate*).
- *Read* (or *Get*): Copies the current record from the buffer to a program variable or work area in the user program. This command may also advance the current record pointer to the next record in the file, which may necessitate reading the next file block from disk.
- *FindNext*: Searches for the next record in the file that satisfies the search condition. Transfers the block containing that record into a main memory buffer (if it is not already there). The record is located in the buffer and becomes the current record.
- *Delete*: Deletes the current record and (eventually) updates the file on disk to reflect the deletion.
- *Modify*: Modifies some field values for the current record and (eventually) updates the file on disk to reflect the modification.
- *Insert*: Inserts a new record in the file by locating the block where the record is to be inserted, transferring that block into a main memory buffer (if it is not already there), writing the record into the buffer, and (eventually) writing the buffer to disk to reflect the insertion.

The preceding are called **record-at-a-time** operations, because each operation applies to a single record. In some file systems, additional **set-at-a-time** higher-level operations may be applied to a file. Examples of these are:

- *FindAll*: Locates *all* the records in the file that satisfy a search condition.
- *FindOrdered*: Retrieves all the records in the file in some specified order.
- *Reorganize*: Starts the reorganization process. As we shall see, some file organizations require periodic reorganization. An example is to reorder the file records by sorting them on a specified field.

Other operations are needed to prepare the file for access (**Open**) and to indicate that we are through using the file (**Close**). The Open operation typically retrieves the file header and prepares memory buffers for subsequent file operations.

At this point, it is worthwhile to note the difference between the terms *file organization* and *access method*. A **file organization** refers to the organization of the data of a file into records, blocks, and access structures; this includes the way records and blocks are placed on the storage medium and interlinked. An **access method**, on the other hand, consists of a group of programs that allow operations—such as those listed earlier—to be applied to a file. In general, it is possible to apply several different access methods to a file organization. Some access methods, though, can be applied only to files organized in certain ways. For example, we cannot apply an indexed access method to a file without an index (see Chapter 5).

Usually, we expect to use some search conditions more than others. Some files may be **static**, meaning that update operations are rarely performed; other, more **volatile** files may change frequently, so update operations are constantly applied to them. A successful file organization should perform as efficiently as possible the operations we expect to *apply frequently* to the file. For example, consider the EMPLOYEE file described earlier, which stores the records for current employees in a company. We expect to insert new records (when employees are hired), delete records (when employees leave the company), and modify records (say, when an employee's salary is changed). Deleting or modifying a record requires a selection condition to identify a particular record or set of records. Retrieving one or more records also requires a selection condition.

If users expect mainly to apply a search condition based on SSN, the designer must choose a file organization that facilitates locating a record given its SSN value. This may involve physically ordering the records by SSN value or defining an index on SSN (see Chapter 5). Suppose that a second application uses the file to generate employee paychecks and requires that paychecks be grouped by department. For this application it is best to store all employee records having the same department value contiguously, packing them into blocks and perhaps ordering them by name within each department. However, this arrangement conflicts with ordering the records by SSN values. If possible, the designer should choose an organization that allows both operations to be done efficiently. Unfortunately, in many cases there may not be an organization that allows all needed operations on a file to be implemented efficiently. Some file organizations make retrieval on certain search conditions very efficient, but concomitantly update becomes expensive. In such cases a compromise must be chosen that takes into account the expected mix of retrieval and update operations.

In the following sections and in Chapter 5, we discuss different methods for organizing records of a file on disk. Several general techniques, such as ordering, hashing, and indexing, are used to create access methods. In addition, various general techniques for handling insertions and deletions work with many file organizations.

4.6 Files of Unordered Records (Heap Files)

In the simplest and most basic type of organization, records are placed in the file in the order in which they are inserted, and new records are inserted at the end of the file. Such an organization is called a **heap** or **pile** file.* This organization is often used with additional access paths, such as the secondary indexes discussed in Chapter 5. It is also used to collect and store data records for future use.

Inserting a new record is *very efficient*: the last disk block of the file is copied into a buffer; the new record is added; and the block is then **rewritten** back to disk. The address of the last file block is kept in the file header. However, searching for a record using any search condition involves a **linear search** through the file block by block—an expensive procedure. If only one record satisfies the search condition, then, on the average, a program will read into memory and search half the file blocks before it finds the record. For a file of b blocks, this requires searching $(b/2)$ blocks, on average. If no records or several records satisfy the search condition, the program must read and search all b blocks in the file.

To delete a record, a program must first find it, copy the block into a buffer, then delete the record from the buffer, and finally **rewrite the block** back to the disk. This leaves extra unused space in the disk block. Deleting a large number of records in this way results in wasted storage space. Another technique used for record deletion is to have an extra byte or bit, called a **deletion marker**, stored with each record. A record is deleted by setting the deletion marker to a certain value. A different value of the marker indicates a valid (not deleted) record. Search programs consider only valid records in a block when conducting their search. Both of these deletion techniques require periodic **reorganization** of the file to reclaim the unused space of deleted records. During reorganization, the file blocks are accessed consecutively, and records are packed by removing deleted records. After such a reorganization, the blocks are filled to capacity once more. Another possibility is to use the space of deleted records when inserting new records, although this requires extra bookkeeping to keep track of empty locations.

We can use either spanned or unspanned organization for an unordered file and either fixed-length or variable-length records. Modifying a variable-length record may require deleting the old record and inserting a modified record, because the modified record may not fit in its old space on disk.

To read all records in order of the values of some field, we create a sorted copy of the file. Sorting is an expensive operation for a large disk file, and special **external sorting** techniques are used. A common method is a variation of the merge sort technique. First,

*Some file systems such as Digital Equipment Corporation's VAX RMS (Record Management Services) call this organization a *sequential file*.

the records within each block are sorted. Then sorted blocks are merged to create groups of sorted records, each the size of two blocks. Each such group of sorted records is sometimes called a **run**. Runs of two blocks are then merged to form runs of four blocks, and so on, until the final run is the completely sorted file.

For a file of unordered *fixed-length records* using *unspanned blocks* and *contiguous allocation*, it is straightforward to access any record by its **position** in the file. If the file records are numbered 0, 1, 2, . . . , $r - 1$ and the records in each block are numbered 0, 1, . . . , $bfr - 1$, where bfr is the blocking factor, then the i^{th} record of the file is located in block $\lfloor i/bfr \rfloor$ and is the $(i \bmod bfr)^{\text{th}}$ record in that block. Such a file is often called a **relative file*** because records can easily be accessed by their relative position. Accessing a record by its position does not help locate a record based on a search condition; however, it facilitates the construction of access paths on the file, such as the indexes discussed in Chapter 5.

4.7 Files of Ordered Records (Sorted Files)

We can physically order the records of a file on disk based on the values of one of their fields—called the **ordering field**. This leads to an **ordered** or **sequential** file.** If the ordering field is also a **key field** of the file—a field guaranteed to have a unique value in each record—then the field is also called the **ordering key** for the file. Figure 4.7 shows an ordered file with NAME as the ordering key field (assuming that employees have distinct names).

Ordered records have some advantages over unordered files. First, reading the records in order of the ordering field values becomes extremely efficient, since no sorting is required. Second, finding the next record from the current one in order of the ordering field usually requires no additional block accesses, because the next record is in the same block as the current one (unless the current record is the last one in the block). Third, using a search condition based on the value of an ordering key field results in faster access when the binary search technique is used, which constitutes an improvement over linear searches although it is not often used for disk files.

A **binary search** for disk files can be done on the blocks rather than on the records. Suppose that the file has b blocks numbered 1, 2, . . . , b ; the records are ordered by ascending value of their ordering key field; and we are searching for a record whose ordering key field value is K . Assuming that disk addresses of the file blocks are available in the file header, the binary search can be described by Algorithm 4.1. A binary search usually accesses $\log_2(b)$ blocks, whether the record is found or not—an improvement over linear searches, where, on the average, $(b/2)$ blocks are accessed when the record is found and b blocks are accessed when the record is not found.

*For example, VAX RMS (Record Management Services) calls this organization a *relative file*.

**Some file systems, such as Digital Equipment Corporation's VAX RMS system, use the term *sequential file* to denote the unordered file described in Section 4.6.

	NAME	SSN	BIRTHDATE	JOB	SALARY	SEX
block 1	Aaron, Ed					
	Abbott, Diane					
	Acosta, Marc					
block 2	Adams, John					
	Adams, Robin					
	Akers, Jan					
block 3	Alexander, Ed					
	Alfred, Bob					
	Allen, Sam					
block 4	Allen, Troy					
	Anders, Keith					
	Anderson, Rob					
block 5	Anderson, Zach					
	Angeli, Joe					
	Archer, Sue					
block 6	Arnold, Mack					
	Arnold, Steven					
	Atkins, Timothy					
block $n-1$	Wong, James					
	Wood, Donald					
	Woods, Manny					
block n	Wright, Pam					
	Wyatt, Charles					
	Zimmer, Byron					

Figure 4.7 Some blocks of an ordered (sequential) file of EMPLOYEE records with NAME as the ordering field.

ALGORITHM 4.1 Binary search on an ordering key of a disk file.

```

 $l \leftarrow 1; u \leftarrow b; (* b \text{ is the number of file blocks} *)$ 
while ( $u \geq l$ ) do
  begin  $i \leftarrow (l + u) \text{ div } 2;$ 
  read block  $i$  of the file into the buffer;
  if  $K < \text{ordering key field value of the first record in the block}$ 
  then  $u \leftarrow i - 1$ 
  else if  $K > \text{ordering key field value of the last record in the block}$ 
  then  $l \leftarrow i + 1$ 
  else if the record with ordering key field value =  $K$  is in the buffer
  then goto found
  else goto notfound;
end;
goto notfound;

```

A search criterion involving the conditions $>$, $<$, $\geq$, and $\leq$ on the ordering field is quite efficient, since the physical ordering of records means that all records satisfying the condition are contiguous in the file. For example, referring to Figure 4.7, if the search criterion is $(\text{NAME} < 'F')$ —where $<$ means “alphabetically before”—the records satisfying the search criterion are those from the beginning of the file up to the first record that has a NAME value starting with the letter F.

Ordering does not provide any advantages for random or ordered access of the records based on values of a *nonordering field* of the file. In these cases we do a linear search for random access. To access the records in order based on a nonordering field, it is necessary to create another sorted copy of the file.

Inserting and deleting records are expensive operations for an ordered file because the records must remain physically ordered. To insert a new record, we must find its correct position in the file, based on its ordering field value, and then make space in the file to insert the record in that position. For a large file this can be very time-consuming because, on the average, half the records of the file must be moved to make space for the new record. This means that half the file blocks must be read and rewritten after records are moved among them. For record deletion the problem is less severe, if we use deletion markers and reorganize the file periodically.

One option for making insertion more efficient is to keep some unused space in each block for new records. However, once this space is used up, the original problem resurfaces. Another frequently used method is to create a temporary *unordered* file called an **overflow** or **transaction** file. With this technique, the actual ordered file is called the **main** or **master** file. New records are inserted at the end of the overflow file rather than in their correct position in the main file. Periodically, the overflow file is merged with the master file during file reorganization. Insertion becomes very efficient, but at the cost of increased complexity in the search algorithm. The overflow file must be searched using a linear search if, after the binary search, the record is not found in the main file. For applications that do not require the most up-to-date information, overflow records can be ignored during a search.

Modifying a field value of a record depends on two factors: the search condition to locate the record, and the field to be modified. If the search condition involves the order-

ing key field, we can locate the record using a binary search; otherwise we must do a linear search. A nonordering field can be modified by changing the record and rewriting it in the same physical location on disk—assuming fixed-length records. Modifying the ordering field means that the record can change its position in the file, which requires deletion of the old record followed by insertion of the modified record.

Reading the file records in order of the ordering field is quite efficient if we ignore the records in overflow, since the blocks can be read consecutively using double buffering. To include the records in overflow, we must merge them in their correct positions; in this case, we can first reorganize the file, and then read its blocks sequentially. To reorganize the file, we first sort the records in the overflow file, and then merge them with the master file. The records marked deleted are removed during the reorganization.

Ordered files are rarely used in database applications unless an additional access path, called a *primary index*, is included with the file. This further improves the random access time on the ordering key field. We discuss indexes in Chapter 5.

4.8 Hashing Techniques

Another type of primary file organization is based on hashing, which provides very fast access to records on certain search conditions. This organization is usually called a **hash** or **direct** file.* The search condition must be an equality condition on a single field, called the **hash field** of the file. Often, the hash field is also a key field of the file, in which case it is called the **hash key**. The idea behind hashing is to provide a function h , called a **hash function** or **randomizing function**, that is applied to the hash field value of a record and yields the *address* of the disk block in which the record is stored. A search for the record within the block can be carried out in a main memory buffer. For most records, we need only a single block access to retrieve that record.

Hashing is also used as an internal data structure within a program whenever a small temporary file of records is accessed exclusively by using the value of one field. We describe the use of hashing for internal files in Section 4.8.1; then we show how it is modified to store external files on disk in Section 4.8.2. In Section 4.8.3 we discuss techniques for extending hashing to dynamically growing files.

4.8.1 Internal Hashing

For internal files, hashing is typically implemented through the use of an array of records. Suppose that the array index range is from 0 to $M - 1$ (Figure 4.8(a)); then we have M **slots** whose addresses correspond to the array indexes. We choose a hash function that transforms the hash field value into an integer between 0 and $M - 1$. One common hash function is the $h(K) = K \bmod M$ function, which returns the remainder of an integer hash field value K after division by M ; this value is then used for the record address.

*In the VAX RMS file system of Digital Equipment Corporation, the term *direct access* refers to accessing a relative file by record position.

ALGORITHM 4.2 Two simple hashing algorithms. (a) Applying the mod hash function to a character string. (b) Collision resolution by open addressing.

```
(a)  $temp \leftarrow 1$ ;
    for  $i \leftarrow 1$  to 20 do  $temp \leftarrow temp * code(K[i])$ ;
     $hash\_address \leftarrow temp \bmod M$ ;

(b)  $i \leftarrow hash\_address$ ;
    if location  $i$  is occupied
    then begin  $i \leftarrow (i + 1) \bmod M$ ;
        while  $(i \neq hash\_address)$  and location  $i$  is occupied
        do  $i \leftarrow (i + 1) \bmod M$ ;
        if  $(i = hash\_address)$  then all positions are full
            else  $new\_hash\_address \leftarrow i$ ;
    end;
```

Other hashing functions can be used. One technique, called **folding**, involves applying an arithmetic function such as addition or a logical function such as “exclusive or” to different portions of the hash field value to calculate the hash address. Another technique involves picking some digits of the hash field value—for example, the third, fifth, and eighth digits—to form the hash address. The problem with most hashing functions is that they do not guarantee that distinct values will hash to distinct addresses, because the **hash field space**—the number of possible values a hash field can take—is usually much larger than the **address space**—the number of available addresses for records. The hashing function maps the hash field space to the address space.

A **collision** occurs when the hash field value of a new record that is being inserted hashes to an address that already contains a different record. In this situation, we must insert the new record in some other position, since its hash address is occupied. The process of finding another position is called **collision resolution**. There are numerous methods for collision resolution, including the following:

- *Open addressing*: Proceeding from the occupied position specified by the hash address, the program checks the subsequent positions in order until an unused (empty) position is found. Algorithm 4.2(b) may be used for this purpose.
- *Chaining*: For this method, various overflow locations are kept, usually by extending the array with a number of overflow positions. In addition, a pointer field is added to each record location. A collision is resolved by placing the new record in an unused overflow location and setting the pointer of the occupied hash address location to the address of that overflow location. A linked list of overflow records for each hash address is maintained, as shown in Figure 4.8(b).
- *Multiple hashing*: The program applies a second hash function if the first results in a collision. If another collision results, the program uses open addressing or applies a third hash function and then uses open addressing if necessary.

Each collision resolution method requires its own algorithms for insertion, retrieval, and deletion of records. The algorithms for chaining are the simplest. Deletion algo-

rithms for open addressing are rather tricky. Data structures textbooks discuss internal hashing algorithms in more detail.

The goal of a good hashing function is to distribute the records uniformly over the address space so as to minimize collisions while not leaving many unused locations. Simulation and analysis studies have shown that it is usually best to keep a hash table between 70% and 90% full so that the number of collisions remains low and we do not waste too much space. Hence, if we expect to have r records to store in the table, we should choose M locations for the address space such that (r/M) is between 0.7 and 0.9. It may also be useful to choose a prime number for M , since it has been demonstrated that this distributes the hash addresses better over the address space when the mod hashing function is used. Other hash functions may require M to be a power of 2.

4.8.2 External Hashing

Hashing for disk files is called **external hashing**. To suit the characteristics of disk storage, the target address space is made of **buckets**, each of which holds multiple records. A bucket is either one disk block or a cluster of contiguous blocks. The hashing function maps a key into a relative bucket number, rather than assigning an absolute block address to the bucket. A table maintained in the file header converts the bucket number into the corresponding disk block address, as illustrated in Figure 4.9.

The collision problem is less severe with buckets, because as many records as will fit in a bucket can hash to the same bucket without causing problems. However, we must make provisions for the case where a bucket is filled to capacity and a new record being inserted hashes to that bucket. We can use a variation of chaining in which we maintain a pointer in each bucket to a linked list of overflow records for the bucket, as shown in Figure 4.10. The pointers in the linked list should be **record pointers**, which include both a block address and a relative record position within the block.

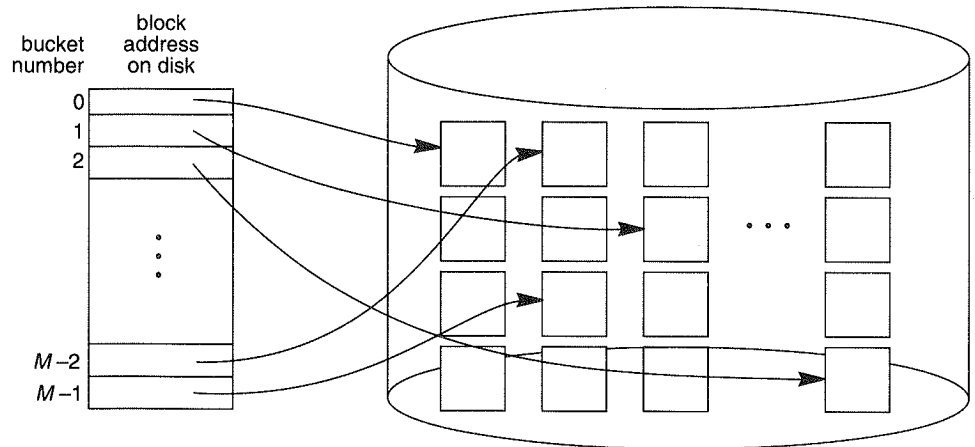


Figure 4.9 Matching bucket numbers to disk blocks.

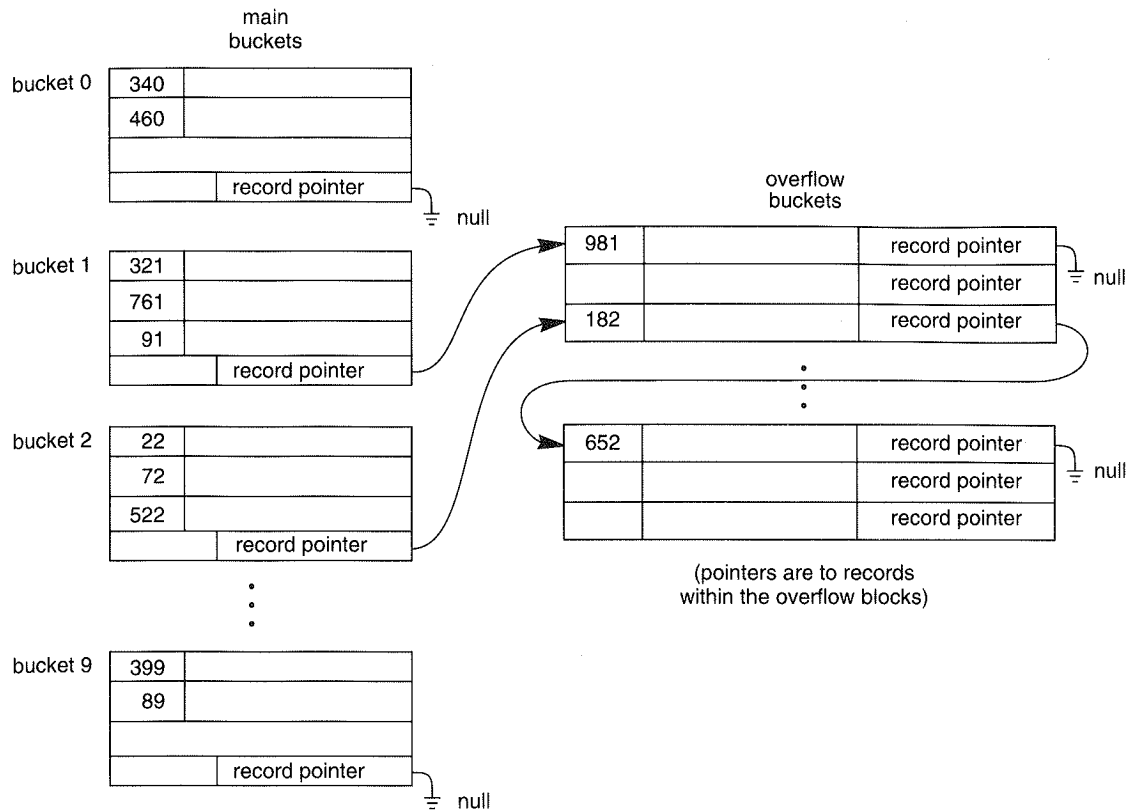


Figure 4.10 Handling overflow for buckets by chaining.

Although hashing provides the fastest possible access for retrieving an arbitrary record, given the value of its hash field, it is not very useful when other applications are required of the same file unless additional access paths are constructed. For example, if we want to retrieve records in order of their hash field values, hashing is not very suitable, since most good hash functions do not maintain records in this order. Some hash functions, called **order preserving**, can maintain the order of records by hash field values. A simple example is to take the leftmost three digits of an invoice number field as the hash address and keep the records sorted by invoice number within each bucket. Another example is to use an integer hash key directly as an index to a relative file, if the hash key values fill up a particular interval; for example, if employee numbers in a company are assigned as 1, 2, 3, ... up to the total number of employees, we can use the identity hash function that maintains order. Unfortunately, this only works if keys are generated in order by some application.

Another drawback of hashing is the fixed amount of space allocated to the file. Suppose that we allocate M buckets for the address space and let m be the maximum number of records that can fit in one bucket; then at most $m * M$ records will fit in the

allocated space. If the number of records turns out to be substantially less than $m * M$, we are left with a lot of unused space. On the other hand, if the number of records increases to substantially more than $m * M$, numerous collisions will result and retrieval will be slowed down because of the long lists of overflow records. In either case, we may have to change the number of blocks allocated and then use a different hashing function to redistribute the records among the buckets. Newer file organizations based on hashing allow the number of buckets to vary dynamically; we discuss some of these techniques in Section 4.8.3.

In regular external hashing, searching for a record given a value of some field other than the hash field is as expensive as in the case of an unordered file. Record deletion can be implemented by removing the record from its bucket. If the bucket has an overflow chain, we can move one of the overflow records into the bucket to replace the deleted record. If the record to be deleted is already in overflow, we simply remove it from the linked list. Notice that removing an overflow record implies that we should keep track of empty positions in overflow. This is easily done by maintaining a linked list of unused overflow locations.

Modifying a record's field value depends on two factors: the search condition to locate the record, and the field to be modified. If the search condition is an equality comparison on the hash field, we can locate the record efficiently by using the hashing function; otherwise, we must do a linear search. A nonhash field can be modified by changing the record and rewriting it in the same bucket. Modifying the hash field means that the record can move to another bucket, which requires deletion of the old record followed by insertion of the modified record.

4.8.3 Hashing Techniques That Allow Dynamic File Expansion*

A major drawback of the *static* hashing scheme just discussed is that the hash address space is fixed. Hence, it is difficult to expand or shrink the file dynamically. The schemes described in this section attempt to remedy this situation. The first two schemes—dynamic hashing and extendible hashing—store an access structure in addition to the file, and hence are somewhat similar to indexing (Chapter 5). The main difference is that the access structure is based on the values that result after application of the hash function to the search field. In indexing, the access structure is based on the value of the search field itself. The third technique—linear hashing—does not require use of any additional access structures.

These hashing schemes take advantage of the fact that the result of most hashing functions is a nonnegative integer and hence can be represented as a binary number. The access structure is built on the **binary representation** of the hashing function result, which is a string of **bits**. We call this the **hash value** of a record. Records are distributed among buckets based on the values of the *leading bits* in their hash values.

Dynamic Hashing. In dynamic hashing, the number of buckets is not fixed (as in regular hashing) but grows or diminishes as needed. The file can start with a single bucket; once that bucket is full, and a new record is inserted, the bucket **overflows** and is split into

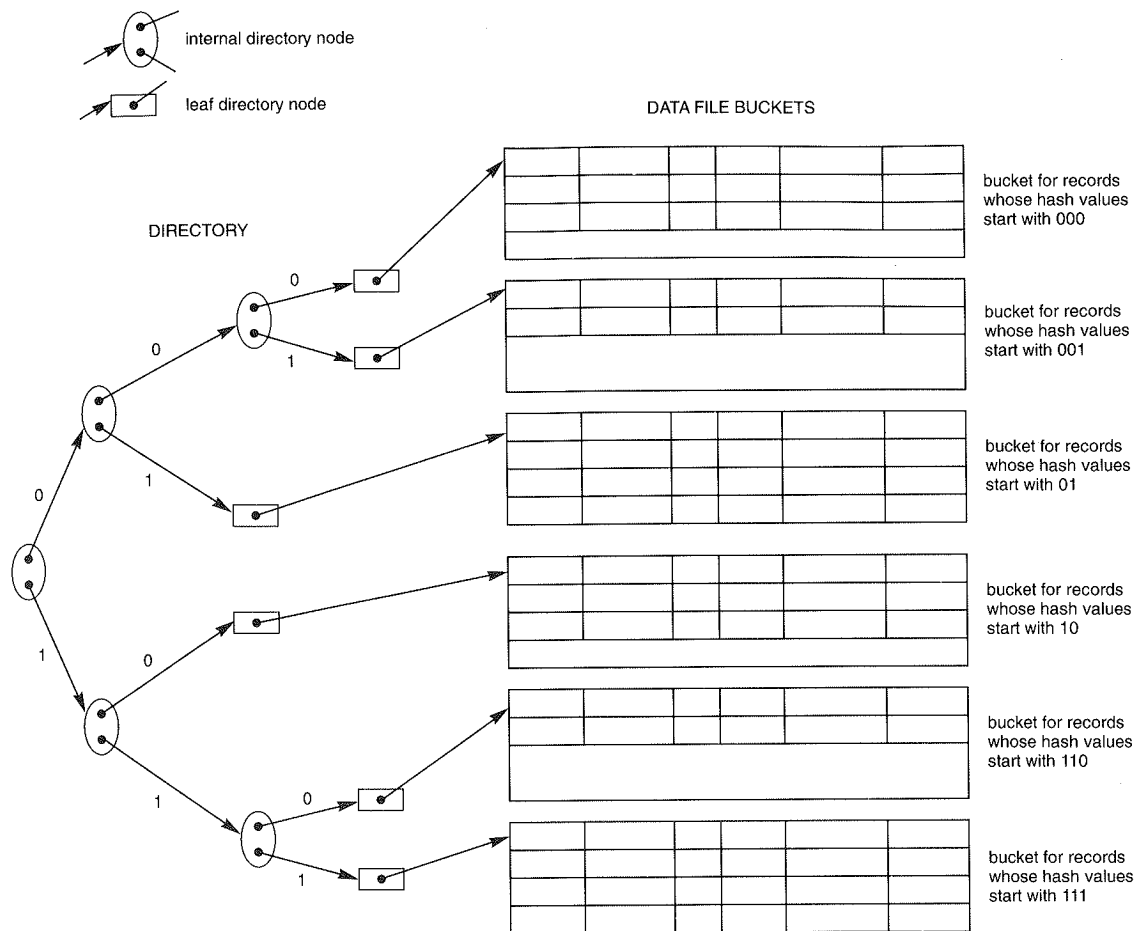


Figure 4.11 Structure of the dynamic hashing scheme.

two buckets. The records are distributed among the two buckets based on the value of the first (leftmost) bit of their hash values. Records whose hash values start with a 0 bit are stored in one bucket, and those whose hash values start with a 1 bit are stored in the other bucket. At this point, a binary tree structure called a **directory** (or **index**) is built. The directory has two types of nodes:

- **Internal nodes** guide the search; each has a left pointer corresponding to a 0 bit and a right pointer corresponding to a 1 bit.
- **Leaf nodes** hold a pointer to a bucket—a bucket address. Figure 4.11 illustrates a directory and the buckets of the data file.

ALGORITHM 4.3 The search procedure for dynamic hashing.

```

 $h \leftarrow$  hash value of record;
 $t \leftarrow$  root node of directory;
 $i \leftarrow 1$ ;
while  $t$  is an internal node of the directory do
  begin
    if the  $i^{\text{th}}$  bit of  $h$  is a 0
      then  $t \leftarrow$  left son of  $t$ 
      else  $t \leftarrow$  right son of  $t$ ;
     $i \leftarrow i + 1$ 
  end;
search the bucket whose address is in node  $t$ ;

```

The search for a record proceeds as shown in Algorithm 4.3. The directory can be stored in main memory unless it becomes too large. If the directory does not fit in one block, it is distributed over two or more levels. Note that directory entries are quite compact. Each internal node holds a tag bit to specify the type of node, plus the left and right pointers. A parent pointer may also be needed. Each leaf node holds a bucket address. Special representations of binary trees can be used to reduce the space needed by left, right, and parent pointers of internal nodes. In general, if a directory of x levels is stored on disk, we need $x + 1$ block accesses to retrieve a bucket.

If a bucket overflows, it is split into two, and the records are distributed based on the next significant bit in their hash value. For example, if a new record is inserted into the bucket for records whose hash values start with 10—the fourth bucket in Figure 4.11—and causes overflow, then all records whose hash value starts with 100 are placed in the first split bucket, and the second bucket contains those whose hash value starts with 101. The directory is expanded with a new internal node to reflect the split; this node points to two leaf nodes that point to the two buckets. The levels of the binary tree can then expand dynamically. However, the number of levels cannot exceed the number of bits in the hash value.

If the hash function distributes the records uniformly, the directory tree will be balanced. Buckets can be combined if one becomes empty or if the total number of records in two neighboring buckets can fit in a single bucket. In this case, the directory loses an internal node and the two leaf nodes are combined to form a single leaf node that points to the new bucket. Thus the levels of the binary tree can shrink dynamically.

Extendible Hashing. In extendible hashing, a different type of **directory**—an array of 2^d bucket addresses—is maintained, where d is called the **global depth** of the directory. The integer value corresponding to the first (high-order) d bits of a hash value is used as an index to the array to determine a directory entry, and the address in that entry determines the bucket in which the corresponding records are stored. However, there does not have to be a distinct bucket for each of the 2^d directory locations. Several directory locations with the same first d bits for their hash values may contain the same bucket address if all the records that hash to these locations fit in a single bucket. A

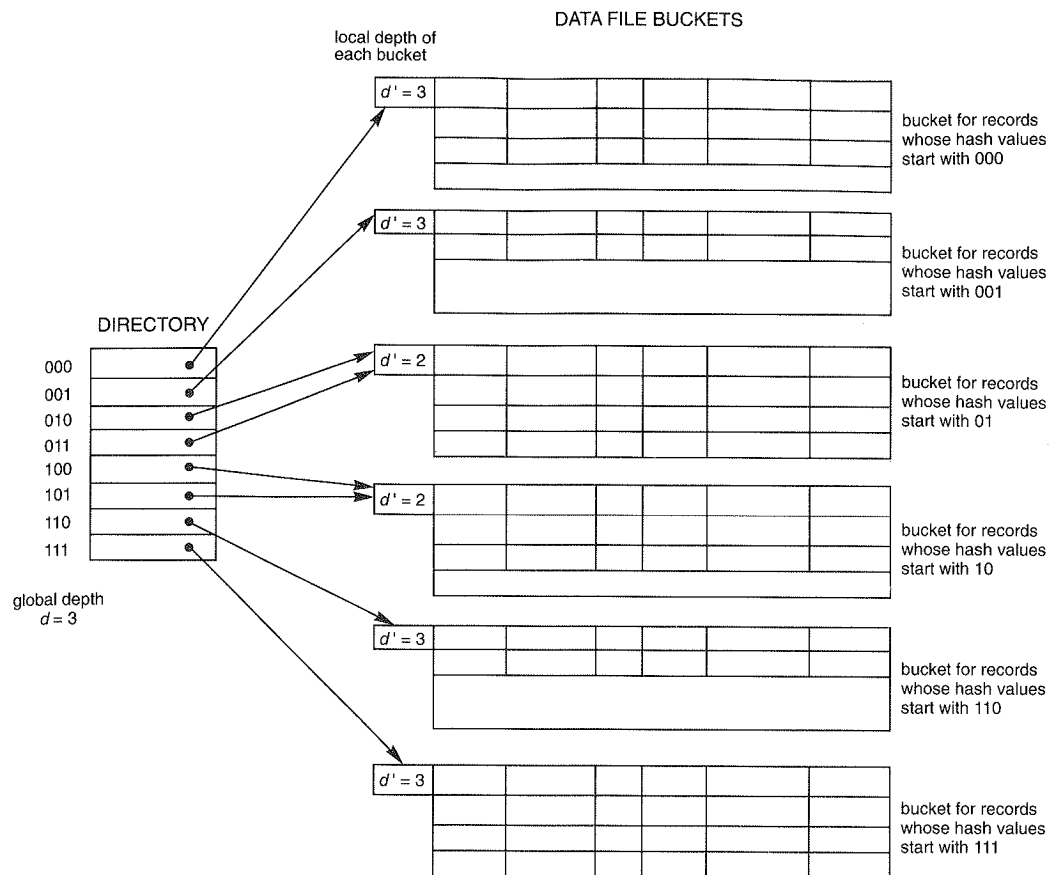


Figure 4.12 Structure of the extendible hashing scheme.

local depth d' —stored with each bucket—specifies the number of bits on which the bucket contents are based. Figure 4.12 shows a directory with global depth $d = 3$.

The value of d can be increased or decreased by one at a time, thus doubling or halving the number of entries in the directory array. Doubling is needed if a bucket whose local depth d' is equal to the global depth d overflows. Halving occurs if $d > d'$ for all the buckets after some deletions occur. Most record retrievals require two block accesses—one to the directory, and the other to the bucket.

To illustrate bucket splitting, suppose that a new inserted record causes overflow in the bucket whose hash values start with 01—the third bucket in Figure 4.12. The records will be distributed among two buckets: the first contains all records whose hash values start with 010, and the second all those whose hash values start with 011. Now the two directory locations for 010 and 011 point to the two new distinct buckets. Before the split, they pointed to the same bucket. The local depth d' of the two new buckets is 3, which is one more than the local depth of the old bucket.

If a bucket that overflows and is split used to have a local depth d' equal to the global depth d of the directory, then the size of the directory must now be doubled so that we can use an extra bit to distinguish the two new buckets. For example, if the bucket for records whose hash values start with 111 in Figure 4.12 overflows, the two new buckets need a directory with global depth $d = 4$, because the two buckets are now labeled 1110 and 1111, and hence their local depths are both 4. The directory size is hence doubled, and each of the other original locations in the directory is also split into two locations, both of which have the same pointer as did the original location. The maximum directory size is 2^k , where k is the number of bits in a hash value.

Linear Hashing. The idea behind linear hashing is to allow a hash file to expand and shrink its number of buckets dynamically *without* needing a directory. Suppose that the file starts with M buckets numbered $0, 1, \dots, M - 1$ and uses the mod hash function $h(K) = K \bmod M$; this hash function is called the initial hash function h_0 . Overflow because of collisions is still handled by maintaining individual overflow chains for each bucket. However, when a collision leads to an overflow record in *any* file bucket, the *first* bucket in the file—bucket 0—is split into two buckets: the original bucket 0 and a new bucket M at the end of the file. The records originally in bucket 0 are distributed among the two buckets based on a different hashing function $h_1(K) = K \bmod 2M$. A key property of the two hash functions h_0 and h_1 is that any records that hashed to bucket 0 based on h_0 will hash to either bucket 0 or bucket M based on h_1 ; this is necessary for linear hashing to work.

As further collisions leading to overflow records occur, additional buckets are split in the *linear* order $1, 2, 3, \dots$. If enough overflows occur, all file buckets are split, so the records in overflow are redistributed into regular buckets, using the function h_1 via a *delayed split* of their buckets. There is no directory; only a value n is needed to determine which buckets have been split. To retrieve a record with hash key value K , first apply the function h_0 to K ; if $h_0(K) < n$, then apply the function h_1 on K because the bucket is already split. Initially, $n = 0$, indicating that the function h_0 applies to all buckets; n grows linearly as buckets are split.

When $n = M$, all the original buckets have been split and the hash function h_1 applies to all records in the file. At this point, n is reset to 0 (zero), and any new collisions causing overflow lead to the use of a new hashing function $h_2(K) = K \bmod 4M$. In general, a sequence of hashing functions $h_j(K) = K \bmod (2^j M)$ is used, where $j = 0, 1, 2, \dots$; a new hashing function h_{j+1} is needed whenever all the buckets $0, 1, \dots, (2^j M) - 1$ have been split and n is reset to 0. The search for a record with hash key value K is given by Algorithm 4.4.

Buckets that have been split can also be recombined if the load of the file falls below a certain threshold. In general, the **file load factor** l can be defined as $l = r / (\text{bfr} * N)$, where r is the current number of file records, bfr is the maximum number of records that can fit in a bucket, and N is the current number of file buckets. Blocks are combined linearly, and N is decremented appropriately. The file load can be used to trigger both splits and combinations; in this manner the file load can be kept within a desired range. Splits can be triggered when the load exceeds a certain threshold—say, 0.9—and combinations can be triggered when the load falls below another threshold—say, 0.7.

ALGORITHM 4.4 The search procedure for linear hashing.

```

if  $n = 0$ 
  then  $m \leftarrow h_j(K)$  (*  $m$  is the hash value of record with hash key  $K$  *)
  else begin
     $m \leftarrow h_j(K)$ ;
    if  $m < n$  then  $m \leftarrow h_{j+1}(K)$ 
  end;
search the bucket whose hash value is  $m$  (and its overflow, if any);

```

4.9 Other Primary File Organizations*

4.9.1 Files of Mixed Records

The file organizations we have studied so far assume that all records of a particular file are of the same record type. The records could be of EMPLOYEES, PROJECTS, STUDENTS, or DEPARTMENTS, but each file contains records of only one type. In most database applications, we encounter situations in which numerous types of entities are interrelated in various ways, as we saw in Chapter 3. Relationships among records in various files can be represented by **connecting fields**. For example, a STUDENT record can have a connecting field MAJORDEPT whose value gives the name of the DEPARTMENT in which the student is majoring. This MAJORDEPT field *refers* to a DEPARTMENT entity, which should be represented by a record of its own in the DEPARTMENT file. If we want to retrieve field values from two related records, we must retrieve one of the records first. Then we can use its connecting field value to retrieve the related record in the other file. Hence, relationships are implemented by **logical field references** among the records in distinct files.

File organizations in hierarchical and network DBMSs implement relationships among records as **physical relationships** realized by physical contiguity of related records or by physical pointers. These file organizations typically assign an **area** of the disk to hold records of more than one type so that records of different record types can be physically related. If a particular relationship is expected to be used very frequently, implementing the relationship physically can increase the system's efficiency at retrieving related records. For example, if the query to retrieve a DEPARTMENT record and all records for STUDENTS majoring in that department is very frequent, it would be desirable to place each DEPARTMENT record and its cluster of STUDENT records contiguously on disk in a mixed file.

To distinguish the records in a mixed file, each record has—in addition to its field values—a **record type** field, which specifies the type of record. This is typically the first field in each record and is used by the system software to determine the type of record it is about to process. Using the catalog information, the DBMS can determine the fields of that record type and their sizes, in order to interpret the record data.

4.9.2 B-Trees and Other Data Structures

Other data structures can be used for primary file organizations. For example, if both the record size and the number of records in a file are small, some DBMSs offer the option of using a B-tree data structure as the primary file organization. We will describe B-trees in Section 5.3.1, when we discuss the use of the B-tree data structure for indexing. In general, any data structure that can be adapted to the characteristics of disk devices can be used as a primary file organization for record placement on disk.

4.10 Summary

In this chapter we started by discussing the characteristics of secondary storage devices. We concentrated on magnetic disks because they are used most often to store on-line database files. Data on disk is stored in blocks; accessing a disk block is expensive because of the seek time, rotational delay, and block transfer time. Double buffering can be used when accessing consecutive disk blocks, to reduce the average block access time. Other disk parameters are discussed in Appendix B.

We discussed different ways of storing records of a file on disk. Records of a file are grouped into disk blocks and can be of fixed length or variable length, spanned or unspanned, and same record type or mixed. We discussed the file header, which describes the record formats and keeps track of the disk addresses of the file blocks. Information in the file header is used by system software accessing the file records.

We then presented a set of typical commands for accessing individual file records and discussed the concept of the current record of a file. We discussed how complex record search conditions are transformed into simple search conditions that are used to locate records in the file.

Three primary file organizations were then discussed: unordered, ordered, and hashed. Unordered files require a linear search to locate records, but record insertion is very simple. We discussed the deletion problem and the use of deletion markers.

Ordered files shorten the time required to read records in order of the ordering field. The time required to search for an arbitrary record, given the value of its ordering key field, is also reduced if a binary search is used. However, maintaining the records in order makes insertion very expensive; thus the technique of using an unordered overflow file to reduce the cost of record insertion was discussed. Overflow records are merged with the master file periodically during file reorganization.

Hashing provides very fast access to an arbitrary record of a file, given the value of its hash field. The most suitable method for external hashing is the bucket technique, with one or more contiguous blocks corresponding to each bucket. Collisions causing bucket overflow are handled by chaining. Access on any nonhash field is slow, and so is sequential access of the records on any field. We then discussed hashing techniques that allow the file to expand and shrink dynamically, including dynamic, extendible, and linear hashing.

Finally, we briefly discussed other possibilities for primary file organizations, such as B-trees, and files of mixed records, which implement relationships among records of different types physically as part of the storage structure.

Review Questions

- 4.1. What is the difference between primary and secondary storage?
- 4.2. Why are disks and not tapes used to store on-line database files?
- 4.3. Define the following terms: *disk*, *disk pack*, *track*, *block*, *cylinder*, *sector*, *interblock gap*, *read/write head*.
- 4.4. Discuss the process of disk initialization.
- 4.5. Discuss the mechanism used to read data from or write data to the disk.
- 4.6. What are the components of a disk block address?
- 4.7. Why is accessing a disk block expensive? Discuss the time components involved in accessing a disk block.
- 4.8. How does double buffering improve block access time?
- 4.9. What are the different reasons for having variable-length records? What type of separator characters is needed for each?
- 4.10. Discuss the different techniques for allocating file blocks on disk.
- 4.11. What is the difference between a file organization and an access method?
- 4.12. What is the difference between a selection condition and a search condition?
- 4.13. What are the typical record-at-a-time operations for accessing a file? Which of these depend on the current record of a file?
- 4.14. Discuss the different techniques for record deletion.
- 4.15. Discuss the advantages and disadvantages of using (a) an unordered file, (b) an ordered file, and (c) a regular (static) hash file with buckets and chaining. Which operations can be efficiently performed on each of these organizations, and which operations are expensive?
- 4.16. Discuss the different techniques for allowing a hash file to expand and shrink dynamically. What are the advantages and disadvantages of each?
- 4.17. What are mixed files used for? What are other types of primary file organizations?

Exercises

- 4.18. Consider a disk with the following characteristics (these are not parameters of any particular disk unit): block size $B = 512$ bytes; interblock gap size $G = 128$ bytes; number of blocks per track = 20; number of tracks per surface = 400. A disk pack consists of 15 double-sided disks.
 - a. What is the total capacity of a track, and what is its useful capacity (excluding interblock gaps)?
 - b. How many cylinders are there?
 - c. What are the total capacity and the useful capacity of a cylinder?
 - d. What are the total capacity and the useful capacity of a disk pack?

- e. Suppose that the disk drive rotates the disk pack at a speed of 2400 rpm (revolutions per minute); what are the transfer rate in bytes/msec and the block transfer time (btt) in msec? What is the average rotational delay (rd) in msec? What is the bulk transfer rate (see Appendix B)?
 - f. Suppose that the average seek time is 30 msec. How much time does it take (on the average) in msec to locate and transfer a single block, given its block address?
 - g. Calculate the average time it would take to transfer 20 random blocks, and compare this with the time it would take to transfer 20 consecutive blocks using double buffering to save seek time and rotational delay.
- 4.19. A file has $r = 20,000$ STUDENT records of *fixed length*. Each record has the following fields: NAME (30 bytes), SSN (9 bytes), ADDRESS (40 bytes), PHONE (9 bytes), BIRTHDATE (8 bytes), SEX (1 byte), MAJORDEPTCODE (4 bytes), MINORDEPTCODE (4 bytes), CLASSCODE (4 bytes, integer), and DEGREEPROGRAM (3 bytes). An additional byte is used as a deletion marker. The file is stored on the disk whose parameters are given in Exercise 4.18.
- a. Calculate the record size R in bytes.
 - b. Calculate the blocking factor bfr and the number of file blocks b , assuming an unspanned organization.
 - c. Calculate the average time it takes to find a record by doing a linear search on the file if (i) the file blocks are stored contiguously, and double buffering is used; (ii) the file blocks are not stored contiguously.
 - d. Assume that the file is ordered by SSN; calculate the time it takes to search for a record given its SSN value, by doing a binary search.
- 4.20. Suppose that only 80% of the STUDENT records from Exercise 4.19 have a value for PHONE, 85% for MAJORDEPTCODE, 15% for MINORDEPTCODE, and 90% for DEGREEPROGRAM; and suppose that we use a variable-length record file. Each record has a 1-byte *field type* for each field occurring in the record, plus the 1-byte deletion marker and a 1-byte end-of-record marker. Suppose that we use a *spanned* record organization, where each block has a 5-byte pointer to the next block (this space is not used for record storage).
- a. Calculate the average record length R in bytes.
 - b. Calculate the number of blocks needed for the file.
- 4.21. Suppose that a disk unit has the following parameters: seek time $s = 20$ msec; rotational delay $rd = 10$ msec; block transfer time $btt = 1$ msec; block size $B = 2400$ bytes; interblock gap size $G = 600$ bytes. An EMPLOYEE file has the following fields: SSN, 9 bytes; LASTNAME, 20 bytes; FIRSTNAME, 20 bytes; MIDDLE INIT, 1 byte; BIRTHDATE, 10 bytes; ADDRESS, 35 bytes; PHONE, 12 bytes; SUPERVISORSSN, 9 bytes; DEPARTMENT, 4 bytes; JOBCODE, 4 bytes; *deletion marker*, 1 byte. The EMPLOYEE file has $r = 30000$ records, fixed-length format, and unspanned blocking. Write down appropriate formulas *and* calculate the following values for the above EMPLOYEE file:

- a. The record size R (including the deletion marker), the blocking factor Bfr , and the number of disk blocks b .
 - b. Calculate the wasted space in each disk block because of the unspanned organization.
 - c. Calculate the transfer rate tr and the bulk transfer rate btr for this disk unit (see Appendix B for definitions of tr and btr).
 - d. Calculate the average *number of block accesses* needed to search for an arbitrary record in the file, using linear search.
 - e. Calculate the average *time* needed in msec to search for an arbitrary record in the file, using linear search, if the file blocks are stored on consecutive disk blocks and double buffering is used.
 - f. Calculate the average *time* needed in msec to search for an arbitrary record in the file, using linear search, if the file blocks are *not* stored on consecutive disk blocks.
 - g. Assume that the records are ordered via some key field. Calculate the average *number of block accesses* and the *average time* needed to search for an arbitrary record in the file, using binary search.
- 4.22. A PARTS file with Part# as hash key includes records with the following Part# values: 2369, 3760, 4692, 4871, 5659, 1821, 1074, 7115, 1620, 2428, 3943, 4750, 6975, 4981, 9208. The file uses eight buckets, numbered 0 to 7. Each bucket is one disk block and holds two records. Load these records into the file in the given order, using the hash function $h(K) = K \bmod 8$. Calculate the average number of block accesses for a random retrieval on Part#.
 - 4.23. Load the records of Exercise 4.22 into expandable hash files based on (i) dynamic hashing and (ii) extendible hashing. Show the structure of the directory at each step. For extendible hashing, show the global and local depths at each stage. Use the hash function $h(K) = K \bmod 32$.
 - 4.24. Load the records of Exercise 4.22 into an expandable hash file, using linear hashing. Start with a single disk block, using the hash function $h_0 = K \bmod 2^0$, and show how the file grows and how the hash functions change as the records are inserted. Assume that blocks are split whenever an overflow occurs, and show the value of n at each stage.
 - 4.25. Compare the file commands listed in Section 4.5 to those available on a file access method you are familiar with.
 - 4.26. Suppose that we have an unordered file of fixed-length records that uses an unspanned record organization. Outline algorithms for insertion, deletion, and modification of a file record. State any assumptions you make.
 - 4.27. Suppose that we have an ordered file of fixed-length records and an unordered overflow file to handle insertion. Both files use unspanned records. Outline algorithms for insertion, deletion, and modification of a file record and for reorganizing the file. State any assumptions you make.
 - 4.28. Can you think of techniques other than an unordered overflow file that can be used to make insertion in an ordered file more efficient?

- 4.29. Suppose that we have a hash file of fixed-length records, and suppose that overflow is handled by chaining. Outline algorithms for insertion, deletion, and modification of a file record. State any assumptions you make.
- 4.30. Can you think of techniques other than chaining to handle bucket overflow in external hashing?
- 4.31. Write program code that can be used to access individual fields of records under each of the following circumstances. For each case, state the assumptions you make concerning pointers, separator characters, etc. Determine the type of information needed in the file header in order for your code to be general in each case.
 - a. Fixed-length records with unspanned blocking.
 - b. Fixed-length records with spanned blocking.
 - c. Variable-length records with variable-length fields and spanned blocking.
 - d. Variable-length records with repeating groups and spanned blocking.
 - e. Variable-length records with optional fields and spanned blocking.
 - f. Variable-length records that allow all three cases in parts c, d, and e.

Selected Bibliography

Wiederhold (1983) has a detailed discussion and analysis of secondary storage devices and file organizations. Optical disks are described in Berg and Roth (1989) and analyzed in Ford and Christodoulakis (1991). Other textbooks, listed in the bibliography at the end of Chapters 1 and 2, include discussions of the material presented here. Most data structures textbooks, including Knuth (1973), discuss static hashing in more detail; Knuth has a complete discussion of hash functions and collision resolution techniques, as well as of their performance comparison. Knuth also offers a detailed discussion of techniques for sorting external files. Salzberg et al. (1991) describes a distributed external sorting algorithm.

Morris (1968) is an early paper on hashing. Dynamic hashing is due to Larson (1978), and extendible hashing is described in Fagin et al. (1979). Linear hashing is described by Litwin (1980). There are many proposed variations for dynamic, extendible, and linear hashing; for examples, see Cesarini and Soda (1991), Du and Tong (1991), and Hachem and Berra (1992).

Several textbooks have appeared whose main topic is file organizations and access methods: Smith and Barnes (1987), Salzberg (1988), Miller (1987), and Livadas (1989).

Transaction Processing Concepts

In this chapter we present the concept of an atomic transaction; this is used to represent a logical unit of database processing. We discuss the concurrency control problem, which occurs when multiple transactions submitted by various users interfere with one another in a way that produces incorrect results. We also discuss recovery from transaction failures. Section 17.1 informally discusses why concurrency control and recovery are necessary in a database system. Section 17.2 introduces the concept of an **atomic transaction** and discusses additional concepts related to transaction processing in database systems. Section 17.3 presents the concepts of atomicity, consistency, isolation, and durability—the so-called **ACID properties** that are considered desirable in transactions. Section 17.4 introduces the concept of **schedules** (or **histories**) of executing transactions and characterizes the recoverability of schedules. Section 17.5 discusses the concept of serializability of concurrent transaction executions, which can be used to define correct execution sequences of concurrent transactions. In Chapter 18 we discuss concurrency control techniques. Chapter 19 presents an overview of recovery techniques.

17.1 Introduction to Transaction Processing

In this section we informally introduce the concepts of concurrent execution of transactions and recovery from transaction failures. In Section 17.1.1 we compare single-user and multiuser database systems and show how concurrent execution of transactions can

take place in multiuser systems. Section 17.1.2 presents a simple model of transaction execution, based on read and write database operations, that is used to formalize concurrency control and recovery concepts. Section 17.1.3 shows by informal examples why concurrency control techniques are needed in multiuser systems. Finally, Section 17.1.4 discusses why techniques are needed to permit recovery from failure by presenting the different ways in which transactions can fail while executing.

17.1.1 Single-User Versus Multiuser Systems

One criterion for classifying a database system is by the number of users who can use the system *concurrently*—that is, at the same time. A DBMS is **single-user** if at most one user at a time can use the system, and it is **multiuser** if many users can use the system concurrently. Single-user DBMSs are mostly restricted to some microcomputer systems; most other DBMSs are multiuser. For example, an airline reservations system is used by hundreds of travel agents and reservation clerks concurrently. Systems in banks, insurance agencies, stock exchanges, and the like are also operated by many users who submit transactions concurrently to the system.

Multiple users can use computer systems simultaneously because of the concept of **multiprogramming**, which allows the computer to process multiple programs (or transactions) at the same time. If only a single central processing unit (CPU) exists, it can actually process at most one program at a time. However, **multiprogramming operating systems** execute some commands from one program, then suspend that program and execute some commands from the next program, and so on. A program is resumed at the point where it was suspended when it gets its turn to use the CPU again. Hence, concurrent execution of the programs is actually **interleaved**, as illustrated in Figure 17.1 (same as Figure 4.3), which shows two programs A and B executing concurrently in an interleaved fashion. Interleaving keeps the CPU busy when an executing program requires an input or output (I/O) operation, such as reading a block from disk. The CPU is switched to execute another program rather than remaining idle during I/O time.

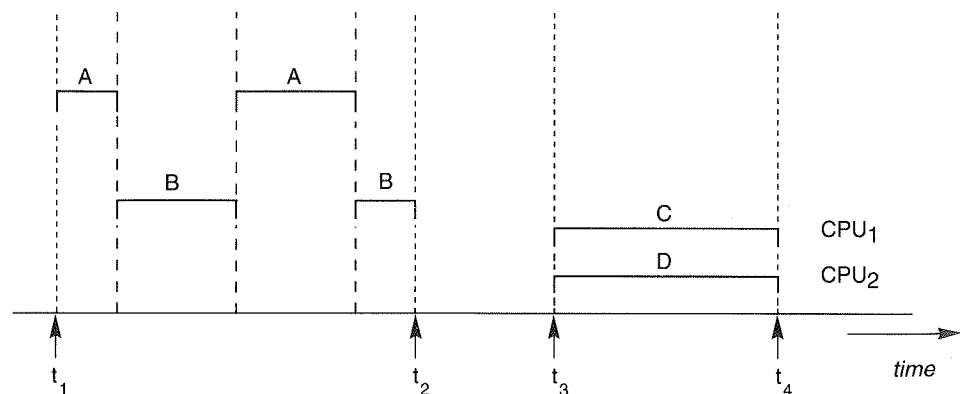


Figure 17.1 Interleaved versus simultaneous concurrency.

If the computer system has multiple hardware processors (CPUs), **simultaneous** processing of multiple programs is possible, leading to simultaneous rather than interleaved concurrency, as illustrated by programs C and D in Figure 17.1. Most of the theory concerning concurrency control in databases is developed in terms of interleaved concurrency, although it may be adapted to simultaneous concurrency. For the remainder of this chapter, we assume the *interleaved model of concurrent execution*.

In a multiuser DBMS, the stored data items are the primary resources that may be accessed concurrently by user programs, which are constantly retrieving information from and modifying the database. The *execution of a program* that accesses or changes the contents of the database is called a **transaction**.

17.1.2 Read and Write Operations of a Transaction

We deal with transactions at the level of data items and disk blocks for the purpose of discussing concurrency control and recovery techniques. At this level, the database access operations that a transaction can include are:

- **read_item(X)**: Reads a database item named X into a program variable. To simplify our notation, we assume that *the program variable is also named X*.
- **write_item(X)**: Writes the value of program variable X into the database item named X.

As we discussed in Chapter 4, the basic unit of data transfer from the disk to the computer main memory is one block. In general, a data item will be the field of some record in the database, although it may be a larger unit such as a record or even a whole block. Executing a `read_item(X)` command includes the following steps:

1. Find the address of the disk block that contains item X.
2. Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
3. Copy item X from the buffer to the program variable named X.

Executing a `write_item(X)` command includes the following steps:

1. Find the address of the disk block that contains item X.
2. Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
3. Copy item X from the program variable named X into its correct location in the buffer.
4. Store the updated block from the buffer back to disk (either immediately or at some later point in time).

Step 4 is the one that actually updates the database on disk. In some cases the buffer is not immediately stored to disk, in case additional changes are to be made to the buffer. Usually, the decision about when to store back a modified disk block that is in a main memory buffer is handled by the recovery manager or the operating system.

(a)	<code>read_item (X);</code> <code>X:=X-N;</code> <code>write_item (X);</code> <code>read_item (Y);</code> <code>Y:=Y+N;</code> <code>write_item (Y);</code>	(b)	<code>read_item (X);</code> <code>X:=X+M;</code> <code>write_item (X);</code>
-----	--	-----	---

Figure 17.2 Two sample transactions. (a) Transaction T_1 .
(b) Transaction T_2 .

A transaction will include `read_item` and `write_item` operations to access the database. Figure 17.2 shows examples of two very simple transactions. Concurrency control and recovery mechanisms are mainly concerned with the database access commands in a transaction.

Transactions submitted by the various users may execute concurrently and may access and update the same database items. If this concurrent execution is uncontrolled, it may lead to problems such as an *inconsistent database*. In the next section we *informally* introduce three of the problems that may occur when concurrent transactions execute in an uncontrolled manner.

17.1.3 Why Concurrency Control Is Needed

Several problems can occur when concurrent transactions execute in an uncontrolled manner. We illustrate some of these problems by referring to a simple airline reservation database in which a record is stored for each airline flight. Each record includes the number of reserved seats on that flight as a *named data item*, among other information. Figure 17.2(a) shows a transaction T_1 that *cancels* N reservations from one flight whose number of reserved seats is stored in the database item named X and *reserves* the same number of seats on another flight whose number of reserved seats is stored in the database item named Y . Figure 17.2(b) shows a simpler transaction T_2 that just *reserves* M seats on the first flight referenced in transaction T_1 . To simplify our example, we do not show additional portions of the transactions, such as checking whether a flight has enough seats available before reserving additional seats.

When a database program is written, it has the flight numbers, their dates, and the number of seats to be booked as parameters; hence, the same program can be used to execute many transactions, each with different flights and numbers of seats to be booked. For concurrency control purposes, a transaction is a *particular execution* of a program on a specific date, flight, and number of seats. In Figure 17.2(a) and (b), the transactions T_1 and T_2 are *specific executions* of the programs that refer to the specific flights whose numbers of seats are stored in data items X and Y in the database. We now discuss the types of problems we may encounter with these two transactions.

The Lost Update Problem. This occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database item incorrect. Suppose that transactions T_1 and T_2 are submitted at approximately the same time, and suppose that their operations are interleaved by the operating

system, as shown in Figure 17.3(a); then the final value of item X is incorrect, because T_2 reads the value of X *before* T_1 changes it in the database, and hence the *updated value resulting from T_1 is lost*. For example, if $X = 80$ at the start (originally there were 80 reservations on the flight), $N = 5$ (T_1 cancels 5 seats on the flight corresponding to X and reserves them on the flight corresponding to Y), and $M = 4$ (T_2 reserves 4 seats on X), the final result should be $X = 79$; but in the schedule of Figure 17.3(a), it is $X = 84$ because the update that canceled 5 seats was *lost*.

The Temporary Update (or Dirty Read) Problem. This occurs when one transaction updates a database item and then the transaction fails for some reason (see Section 17.1.4). The updated item is accessed by another transaction before it is changed back to its original value. Figure 17.3(b) shows an example where T_1 updates item X and then

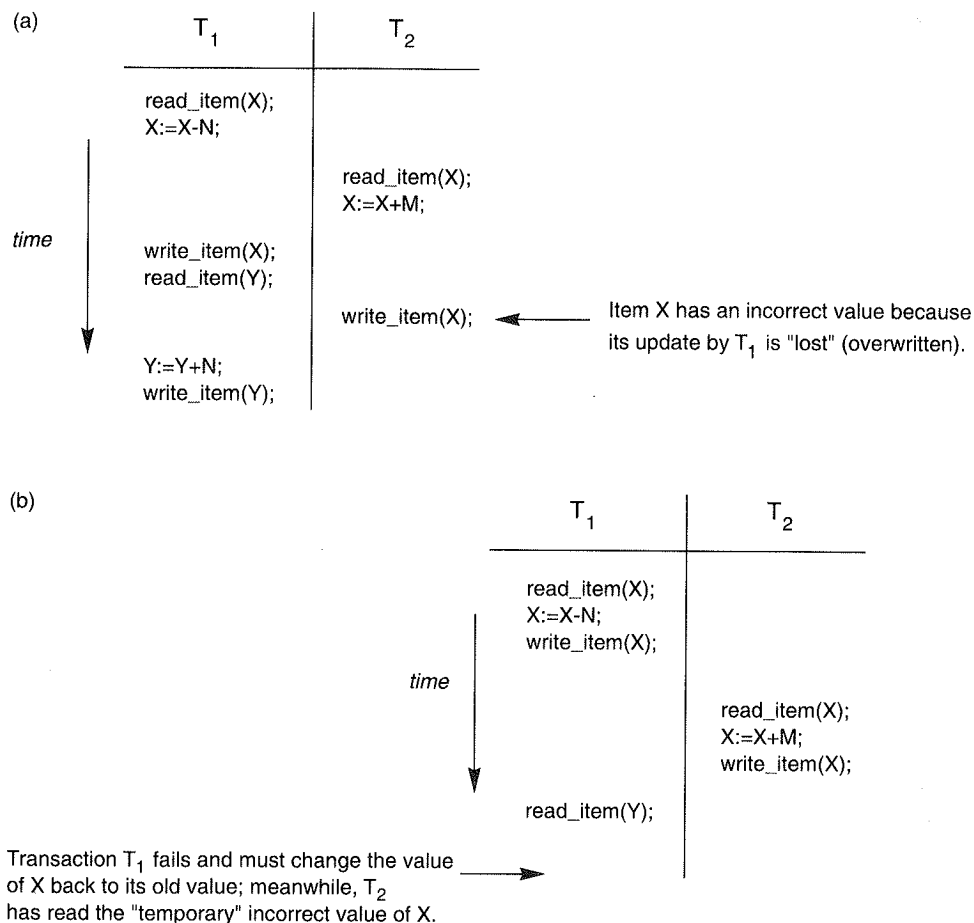


Figure 17.3 Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (continued on next page)

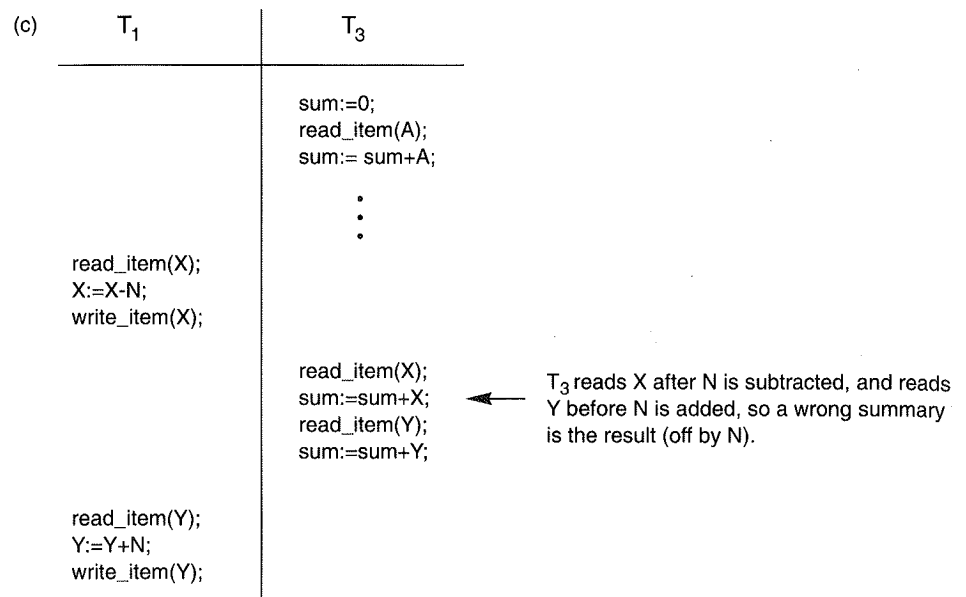


Figure 17.3 (continued) (c) The incorrect summary problem.

fails before completion, so the system must change X back to its original value. Before it can do so, however, transaction T_2 reads the “temporary” value of X, which will not be recorded permanently in the database because of the failure of T_1 . The value of item X that is read by T_2 is called *dirty data*, because it has been created by a transaction that has not completed and committed yet; hence, this problem is also known as the *dirty read* problem.

The Incorrect Summary Problem. If one transaction is calculating an aggregate summary function on a number of records while other transactions are updating some of these records, the aggregate function may calculate some values before they are updated and others after they are updated. For example, suppose that a transaction T_3 is calculating the total number of reservations on all the flights; meanwhile, transaction T_1 is executing. If the interleaving of operations shown in Figure 17.3(c) occurs, the result of T_3 will be off by an amount N because T_3 reads the value of X *after* N seats have been subtracted from it but reads the value of Y *before* those N seats have been added to it.

Another problem that may occur is the **unrepeatable read**, where a transaction T reads an item twice, and the item is changed by another transaction T' between the two reads. Hence, T receives *different values* for its two reads of the same item.

17.1.4 Why Recovery Is Needed

Whenever a transaction is submitted to a DBMS for execution, the system is responsible for making sure that either (a) all the operations in the transaction are completed suc-

cessfully and their effect is recorded permanently in the database, or (b) the transaction has no effect whatsoever on the database or on any other transactions. The DBMS must not permit some operations of a transaction *T* to be applied to the database while other operations of *T* are not. This may happen if a transaction **fails** after executing some of its operations but before executing all of them.

Types of Failures. There are several possible reasons for a transaction to fail in the middle of execution:

1. A computer failure (system crash): A hardware or software error occurs in the computer system during transaction execution. If the hardware crashes, the contents of the computer's internal memory may be lost.
2. A transaction or system error: Some operation in the transaction may cause it to fail, such as integer overflow or division by zero. Transaction failure may also occur because of erroneous parameter values or because of a logical programming error. In addition, the user may interrupt the transaction during its execution on purpose—for example, by issuing a control-C in a VAX/VMS or UNIX environment.
3. Local errors or exception conditions detected by the transaction: During transaction execution, certain conditions may occur that necessitate cancellation of the transaction. For example, data for the transaction may not be found. A condition, such as insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal from that account, to be canceled. This may be done by a programmed ABORT in the transaction itself.
4. Concurrency control enforcement: The concurrency control method may decide to abort the transaction, to be restarted later, because it violates serializability or because several transactions are in a state of deadlock (see Chapter 18).
5. Disk failure: Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash. This may happen during a read or a write operation of the transaction.
6. Physical problems and catastrophes: This refers to an endless list of problems that includes power or air-conditioning failure, fire, theft, sabotage, overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator.

Failures of types 1, 2, 3, and 4 are more common than those of types 5 or 6. Whenever a failure of type 1 through 4 occurs, the system must keep sufficient information to recover from the failure. Disk failure or other catastrophic failures of type 5 or 6 do not happen frequently; if they do occur, it is a major task to recover from these types of failure. We discuss concepts and techniques for recovery from failure in Chapter 19.

The concept of an atomic transaction is fundamental to many techniques for concurrency control and recovery from failures. In the next section we present the transaction concept and discuss why it is important.

17.2 Transaction and System Concepts

As mentioned earlier, the *execution of a program* that includes database access operations is called a **database transaction** or simply a **transaction**. If the database operations in a transaction do not update any data in the database but only retrieve data, the transaction is called a **read-only transaction**. In this chapter we are mainly interested in transactions that do some updates to the database, so the word *transaction* refers to a program execution that *updates the database* unless we explicitly state otherwise.

17.2.1 Transaction States and Additional Operations

A transaction is an atomic unit of work that is either completed in its entirety or not done at all. For recovery purposes, the system needs to keep track of when the transaction starts, terminates, and commits or aborts (see below). Hence, the recovery manager keeps track of the following operations:

- **BEGIN_TRANSACTION**: This marks the beginning of transaction execution.
- **READ or WRITE**: These specify read or write operations on the database items that are executed as part of a transaction.
- **END_TRANSACTION**: This specifies that **READ** and **WRITE** transaction operations have ended and marks the end limit of transaction execution. However, at this point it may be necessary to check whether the changes introduced by the transaction can be permanently applied to the database (committed) or whether the transaction has to be aborted because it violates concurrency control or for some other reason.
- **COMMIT_TRANSACTION**: This signals a *successful end* of the transaction so that any changes (updates) executed by the transaction can be safely **committed** to the database and will not be undone.
- **ROLLBACK (or ABORT)**: This signals that the transaction has *ended unsuccessfully*, so that any changes or effects that the transaction may have applied to the database must be *undone*.

In addition to the preceding operations, some recovery techniques require additional operations that include the following:

- **UNDO**: Similar to rollback except that it applies to a single operation rather than to a whole transaction.
- **REDO**: This specifies that certain *transaction operations* must be *redone* to ensure that all the operations of a committed transaction have been applied successfully to the database.

Figure 17.4 shows a state transition diagram that describes how a transaction moves through its execution states. A transaction goes into an **active** state immediately after it starts execution, where it can issue **READ** and **WRITE** operations. When the transaction ends, it moves to the **partially committed** state. At this point, some concurrency control techniques require that certain *checks* be made to ensure that the transaction did not

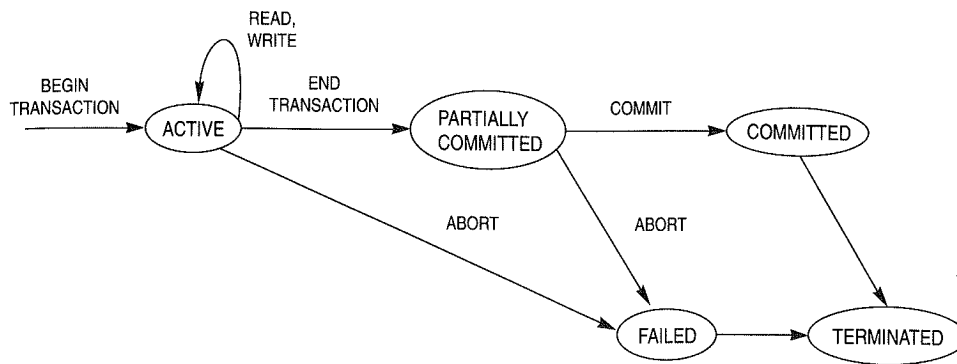


Figure 17.4 State transition diagram for transaction execution.

interfere with other executing transactions. In addition, some recovery protocols need to ensure that a system failure will not result in an inability to record the changes of the transaction permanently (usually by recording changes in a system log, discussed in the next subsection). Once both checks are successful, the transaction is said to have reached its commit point and enters the **committed state**. Commit points are discussed in more detail in Section 17.2.3. Once a transaction enters the committed state, it has concluded its execution successfully.

However, a transaction can go to the **failed** state if one of the checks fails or if it is aborted during its active state. The transaction may then have to be rolled back to undo the effect of its WRITE operations on the database. The **terminated** state corresponds to the transaction leaving the system. Failed or aborted transactions may be *restarted* later, either automatically or after being resubmitted, as *brand new* transactions.

17.2.2 The System Log

To be able to recover from transaction failures, the system maintains a **log** (sometimes called a **journal**). The log keeps track of all transaction operations that affect the values of database items. This information may be needed to permit recovery from transaction failures. The log is kept on disk, so it is not affected by any type of failure except for disk or catastrophic failure. In addition, the log is periodically backed up to archival storage (tape) to guard against such catastrophic failures. We now list the types of entries that are written to the log and the action each performs. In these entries, T refers to a unique **transaction-id** that is generated automatically by the system and is used to identify each transaction:

1. [start_transaction,T]: Records that transaction T has started execution.
2. [write_item,T,X,old_value,new_value]: Records that transaction T has changed the value of database item X from old_value to new_value.
3. [read_item,T,X]: Records that transaction T has read the value of database item X.

4. [commit,T]: Records that transaction T has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.
5. [abort,T]: Records that transaction T has been aborted.

As we shall see in Chapter 19, protocols for recovery that avoid cascading rollbacks do not require that READ operations be written to the system log, whereas other protocols require these entries for recovery. In the former case, the overhead of recording operations in the log is reduced, since fewer operations—only WRITES—are recorded in the log. In addition, strict protocols require simpler WRITE entries that do not include *new_value* (see Section 17.4).

Notice that we assume here that transactions cannot be nested. We also assume that *all* permanent changes to the database occur within transactions, so the notion of recovery from a transaction failure amounts to either undoing or redoing the *recoverable* transaction operations individually from the log. If the system crashes, we can recover to a consistent database state by examining the log and using one of the techniques described in Chapter 19. Because the log contains a record of every WRITE operation that changes the value of some database item, it is possible to **undo** the effect of these WRITE operations of a transaction T by tracing backward through the log and resetting all items changed by a WRITE operation of T to their *old_values*. We can also **redo** the effect of the WRITE operations of a transaction T by tracing forward through the log and setting all items changed by a WRITE operation of T to their *new_values*. Redoing the operations of a transaction may be required if all its updates are recorded in the log but a failure occurs before we can be sure that all the *new_values* have been written permanently in the actual database.

17.2.3 Commit Point of a Transaction

A transaction T reaches its **commit point** when all its operations that access the database have been executed successfully *and* the effect of all the transaction operations on the database has been recorded in the log. Beyond the commit point, the transaction is said to be **committed**, and its effect is assumed to be *permanently recorded* in the database. The transaction then writes an entry [commit,T] into the log. If a system failure occurs, we search back in the log for all transactions T that have written a [start_transaction,T] entry into the log but have not written their [commit,T] entry yet; these transactions may have to be *rolled back* to undo their effect on the database during the recovery process. Transactions that have written their commit entry in the log must also have recorded all their WRITE operations in the log; otherwise they would not be committed, so their effect on the database can be *redone* from the log entries.

Notice that the log file must be kept on disk. As discussed in Chapter 4, updating a disk file involves copying the appropriate block of the file from disk to a buffer in main memory, updating the buffer in main memory, and copying the buffer back from main memory to disk. It is common to keep one block of the log file in main memory until it is filled with log entries and then to write it back to disk only once, rather than writing it to disk every time a log entry is added. This saves the overhead of multiple disk writes of the same file block. At the time of a system crash, only the log entries that have been *written back to disk* are considered in the recovery process because the contents of main

memory may be lost. Hence, *before* a transaction reaches its commit point, any portion of the log that has not been written to the disk yet must now be written to the disk. This process is called **force-writing** the log file before committing a transaction.

17.2.4 Checkpoints in the System Log

Another type of entry in the log is called a **checkpoint**.^{*} A [checkpoint] record is written into the log periodically at that point when the system writes out to the database on disk the effect of all WRITE operations of committed transactions. Hence, all transactions that have their [commit,T] entries in the log before a [checkpoint] entry do not need to have their WRITE operations *redone* in case of a system crash.

The recovery manager of a DBMS must decide at what intervals to take a checkpoint. The interval may be measured in time—say, every *m* minutes—or the number *t* of committed transactions since the last checkpoint, where the values of *m* or *t* are system parameters. Taking a checkpoint consists of the following actions:

1. Suspend execution of transactions temporarily.
2. Force-write all update operations of committed transactions from main memory buffers to disk.
3. Write a [checkpoint] record to the log, and force-write the log to disk.
4. Resume executing transactions.

A checkpoint record in the log may also include additional information, such as a list of active transaction ids, and the locations (addresses) of the first and most recent (last) records in the log for each active transaction. This can facilitate undoing transaction operations in the event that a transaction must be rolled back.

17.3 Desirable Properties of Transactions

Atomic transactions should possess several properties. These are often called the **ACID properties**, and they should be enforced by the concurrency control and recovery methods of the DBMS. The following are the ACID properties:

1. **Atomicity:** A transaction is an atomic unit of processing; it is either performed in its entirety or not performed at all.
2. **Consistency preservation:** A correct execution of the transaction must take the database from one consistent state to another.
3. **Isolation:** A transaction should not make its updates visible to other transactions until it is committed; this property, when enforced strictly, solves the temporary update problem and makes cascading rollbacks of transactions unnecessary (see Chapter 19).

^{*}The term *checkpoint* has been used to describe more restrictive situations in some systems, such as DB2. It has also been used in the literature to describe altogether different concepts.

4. **Durability or permanency:** Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure.

The atomicity property requires that we execute a transaction to completion. It is the responsibility of the recovery method to ensure atomicity. If a transaction fails to complete for some reason, such as a system crash in the midst of transaction execution, the recovery method must undo any effects of the transaction on the database.

The consistency preservation property is generally considered to be the responsibility of the programmers who write the database programs or of the DBMS module that enforces integrity constraints. Recall that a **database state** is a collection of all the stored data items (values) in the database at a given point in time. A **consistent state** of the database satisfies the constraints specified in the schema as well as any other constraints that should hold on the database. A database program should be written in a way that guarantees that, if the database is in a consistent state before executing the transaction, it will be in a consistent state after the *complete* execution of the transaction, assuming that *no interference with other transactions* occurs.

Isolation is enforced by the currency control method. There have been attempts to define the *level* or *degree of isolation* of a transaction. A transaction is said to have degree 0 (zero) isolation if it does not overwrite the dirty reads of higher-degree transactions. A degree 1 (one) isolation transaction has no lost updates; and degree 2 isolation has no lost updates and no dirty reads. Finally, degree 3 isolation (also called *true isolation*) has, in addition to degree 2 properties, repeatable reads. The durability property is the responsibility of the recovery method.

17.4 Schedules and Recoverability

When transactions are executing concurrently in an interleaved fashion, the order of execution of operations from the various transactions forms what is known as a transaction **schedule** (or **history**). In this section, we first define the concept of schedule, and then we characterize the types of schedules that facilitate recovery when failures occur. In Section 17.5, we characterize schedules in terms of the interference of participating transactions, leading to the concepts of serializability and serializable schedules.

17.4.1 Schedules (Histories) of Transactions

A **schedule** (or **history**) S of n transactions $T_1, T_2, \dots, T_n$ is an ordering of the operations of the transactions subject to the constraint that, for each transaction T_i that participates in S , the operations of T_i in S must appear in the same order in which they occur in T_i . Note, however, that operations from other transactions T_j can be interleaved with the operations of T_i in S . For now, consider the order of operations in S to be a *total ordering*, although it is possible to deal with schedules whose operations form *partial orders* (as we discuss later).

Figure 17.3(a) and (b) shows two possible schedules for the transactions T_1 and T_2 . For the purpose of recovery and concurrency control, we are mainly interested in the

read_item and write_item operations of the transactions, as well as the commit and abort operations. A shorthand notation for describing a schedule uses the symbols r , w , c , and a for the operations read_item, write_item, commit, and abort, respectively, and appends as subscript the transaction id (transaction number) to each operation in the schedule. In this notation, the database item X that is read or written follows the r and w operations in parentheses. For example, the schedule of Figure 17.3(a), which we shall call S_a , can be written as follows when augmented with the commit operations:

$S_a: r_1(X); r_2(X); w_1(X); r_1(Y); w_2(X); c_2; w_1(Y); c_1;$

Similarly, the schedule for Figure 17.3(b), which we call S_b , can be written as follows, if we assume that transaction T_1 aborted after its read_item(Y) operation:

$S_b: r_1(X); w_1(X); r_2(X); w_2(X); c_2; r_1(Y); a_1;$

Two operations in a schedule are said to **conflict** if they belong to different transactions, if they access the same item X , and if one of the two operations is a write_item(X). For example, in schedule S_a , the operations $r_1(X)$ and $w_2(X)$ conflict, as do the operations $r_2(X)$ and $w_1(X)$, and the operations $w_1(X)$ and $w_2(X)$. However, the operations $r_1(X)$ and $r_2(X)$ do not conflict, since they are both read operations; and the operations $w_2(X)$ and $w_1(Y)$ do not conflict, because they operate on distinct data items X and Y .

A schedule S of n transactions $T_1, T_2, \dots, T_n$, is said to be a **complete schedule** if the following conditions hold:

1. The operations in S are exactly those operations in $T_1, T_2, \dots, T_n$, including a commit or abort operation as the last operation for each transaction in the schedule.
2. For any pair of operations from the same transaction T_i , their order of appearance in S is the same as their order of appearance in T_i .
3. For any two conflicting operations, one of the two must occur before the other in the schedule.

The preceding conditions allow for two *nonconflicting operations* to occur in the schedule without defining which occurs first, thus leading to the definition of a schedule as a **partial order** of the operations in the n transactions. However, a total order must be specified in the schedule for any pair of conflicting operations (condition 3) and for any pair of operations from the same transaction (condition 2). Condition 1 simply states that all operations in the transactions must appear in the complete schedule. Since every transaction has either committed or aborted, a complete schedule will not contain any active transactions.

In general, it is difficult to encounter complete schedules in a transaction-processing system, because new transactions are continually being submitted to the system. Hence, it is useful to define the concept of the **committed projection** $C(S)$ of a schedule S , which includes only the operations in S that belong to committed transactions—that is, transactions T_i whose commit operation c_i is in S .

17.4.2 Characterizing Schedules Based on Recoverability

For some schedules it is easy to recover from transaction failures, whereas for other schedules the recovery process can be quite involved. Hence, it is important to characterize the types of schedules for which recovery is possible, as well as those for which recovery is relatively simple. First, we would like to ensure that, once a transaction T is committed, it should *never be necessary* to roll back T . The schedules that meet this criterion are called *recoverable schedules*.

A schedule S is said to be **recoverable** if no transaction T in S commits until all transactions T' that have written an item that T reads have committed. A transaction T is said to **read from** transaction T' in a schedule S if some item X is first written by T' and later read by T . In the schedule S , T' should not have aborted before T reads item X , and there should be no transactions that write X after T' writes it and before T reads it (unless those transactions, if any, have aborted before T reads X).

The schedule S_a given in the preceding section is recoverable. However, consider the two (partial) schedules S_c and S_d that follow:

S_c : $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; c_2 ; a_1 ;

S_d : $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$; c_1 ; c_2 ;

S_c is not recoverable, because T_2 reads item X from T_1 , and then T_2 commits before T_1 commits. If T_1 aborts after the c_2 operation in S_c , then the value of X that T_2 read is no longer valid and T_2 must be aborted *after it had already committed*, leading to a schedule that is not recoverable. For the schedule to be recoverable, the c_2 operation in S_c must be postponed until after T_1 commits, as shown in S_d .

In a recoverable schedule, no committed transaction ever needs to be rolled back. However, it is still possible for a phenomenon known as *cascading rollback* (or *cascading abort*) to occur, where an uncommitted transaction has to be rolled back because it read an item from a transaction that failed. This is illustrated in schedule S_e , where transaction T_2 has to be rolled back because it read item X from T_1 , and T_1 then aborted:

S_e : $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$; a_1 ;

Because cascading rollback can be quite time-consuming—since numerous transactions can be rolled back (see Chapter 19)—it is important to characterize the schedules where this phenomenon is guaranteed not to occur. A schedule is said to **avoid cascading rollback** if every transaction in the schedule only reads items that were written by *committed* transactions. In this case, all items read will be committed, so no cascading rollback will occur. To satisfy this criterion, the $r_2(X)$ command in schedule S_e must be postponed until after T_1 has committed (or aborted), thus delaying T_2 but ensuring no cascading rollback if T_1 aborts.

Finally, there is a third, more restrictive type of schedule, called a **strict schedule**, in which transactions can neither read *nor write* an item X until the last transaction that wrote X has committed (or aborted). Strict schedules simplify the process of recovering write operations to a matter of restoring the **before image** of a data item X , which is the value that X had prior to the aborted write operation. This simple procedure may not always work correctly unless a schedule is strict. For example, consider schedule S_f :

S_f : $w_1(X, 5)$; $w_2(X, 8)$; a_1 ;

Suppose that the value of X was originally 9, which is the before image stored in the system log along with the $w_1(X, 5)$ operation. If T_1 aborts, as in S_6 , the recovery procedure that restores the before image of an aborted write operation will restore the value of X to 9, even though it has already been changed to 8 by transaction T_2 , thus leading to potentially incorrect results. Although S_6 avoids cascading aborts, it is not a strict schedule, since it permits T_2 to write item X even though the transaction T_2 that last wrote X had not yet committed (or aborted). A strict schedule does not have this problem; it is also recoverable and avoids cascading rollback.

We have now characterized schedules in terms of their recoverability. In the next section we characterize the type of schedules that are considered correct when concurrent transactions are executing; these are called *serializable schedules*.

17.5 Serializability of Schedules

Suppose that two users—airline reservation clerks—submit to the DBMS transactions T_1 and T_2 of Figure 17.2 at approximately the same time. If no interleaving is permitted, there are only two possible ways of ordering the operations of the two transactions for execution:

1. Execute all the operations of transaction T_1 (in sequence) followed by all the operations of transaction T_2 (in sequence).
2. Execute all the operations of transaction T_2 (in sequence) followed by all the operations of transaction T_1 (in sequence).

These alternatives are shown in Figure 17.5(a) and (b), respectively. If interleaving of operations is allowed, there will be many possible orders in which the system can execute the individual operations of the transactions. Two possible schedules are shown in Figure 17.5(c).

An important aspect of concurrency control, called **serializability theory**, attempts to determine which schedules are “correct” and which are not and to develop techniques that allow only correct schedules. This section defines serializability of schedules, presents some of this theory, and discusses how it may be used in practice.

17.5.1 Serial, Nonserial, and Conflict-Serializable Schedules

Figure 17.5 shows several possible schedules for transactions T_1 and T_2 of Figure 17.2. Schedules A and B in Figure 17.5(a) and (b) are called *serial schedules* because the operations of each transaction are executed consecutively, without any interleaved operations from the other transaction. In a serial schedule, entire transactions are performed in serial order: T_1 and then T_2 in Figure 17.5(a), and T_2 and then T_1 in Figure 17.5(b). Schedules C and D in Figure 17.5(c) are called *nonserial* because each sequence interleaves operations from the two transactions.

Formally, a schedule S is **serial** if, for every transaction T participating in the schedule, all the operations of T are executed consecutively in the schedule; otherwise, the schedule is called **nonserial**. One reasonable assumption we can make, if we consider

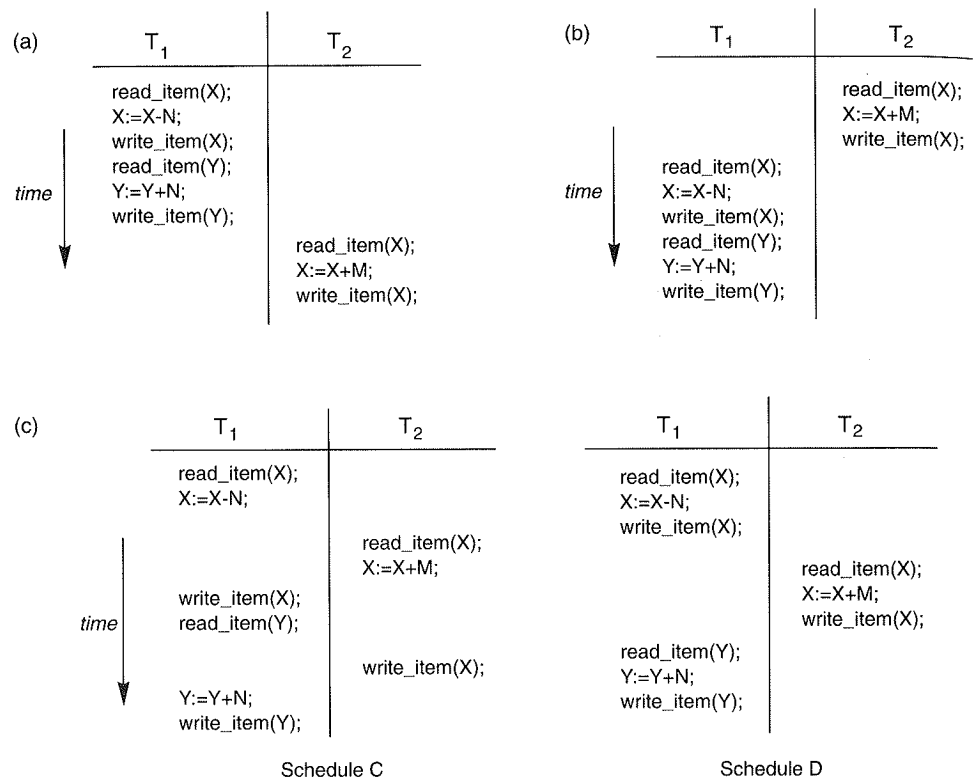


Figure 17.5 Some schedules involving transactions T₁ and T₂. (a) Schedule A: T₁ followed by T₂. (b) Schedule B: T₂ followed by T₁. (c) Two schedules with interleaving of operations.

the transactions to be *independent*, is that *every serial schedule is considered correct*. This is so because we assume that every transaction is correct if executed on its own (by the consistency preservation property of Section 17.3) and that transactions do not depend on one another. Hence, it does not matter which transaction is executed first. As long as every transaction is executed from beginning to end without any interference from the operations of other transactions, we get a correct end result on the database. The problem with serial schedules is that they limit concurrency or interleaving of operations. In a serial schedule, if a transaction waits for an I/O operation to complete, we cannot switch the CPU processor to another transaction, thus wasting valuable CPU processing time and making serial schedules generally unacceptable.

To illustrate our discussion, consider the schedules in Figure 17.5, and assume that the initial values of database items are $X = 90$, $Y = 90$ and that $N = 3$ and $M = 2$. After executing transactions T₁ and T₂, we would expect the database values to be $X = 89$ and $Y = 93$, according to the meaning of the transactions. Sure enough, executing either of the serial schedules A or B gives the correct results. Now consider the nonserial schedules C and D. Schedule C (same as Figure 17.3(a)) gives the results $X = 92$ and $Y = 93$, in which the X value is erroneous, whereas schedule D gives the correct results.

Schedule C gives an erroneous result because of the lost update problem discussed in Section 17.1.3; transaction T_2 reads the value of X *before* it is changed by transaction T_1 , so only the effect of T_2 on X is reflected in the database. The effect of T_1 on X is *lost*, overwritten by T_2 , leading to the incorrect result for item X . However, some non-serial schedules give the correct expected result, such as schedule D. We would like to determine which of the nonserial schedules *always* give a correct result and which may give erroneous results. The concept used to characterize schedules in this manner is that of serializability of a schedule.

A schedule S of n transactions is **serializable** if it is *equivalent to some serial schedule* of the same n transactions. We will define the concept of equivalence of transactions shortly. Notice that there are $(n)!$ possible serial schedules of n transactions and many more possible nonserial schedules. We can form two disjoint groups of the nonserial schedules: those that are equivalent to one (or more) of the serial schedules, and hence are serializable; and those that are not equivalent to *any* serial schedule, and hence are not serializable.

Saying that a nonserial schedule S is serializable is equivalent to saying that it is correct, because it is equivalent to a serial schedule, which is considered correct. The remaining question is: When are two schedules considered “equivalent”? There are several ways to define equivalence of schedules. The simplest, but least satisfactory, definition of schedule equivalence involves comparing the effects of the schedules on the database. Two schedules are called **result equivalent** if they produce the same final state of the database. However, two different schedules may accidentally produce the same final state. For example, in Figure 17.6, schedules S_1 and S_2 will produce the same final database state if they execute on a database with an initial value of $X = 100$; but for other initial values of X , the schedules are *not* result equivalent. In addition, these two schedules execute different transactions, so they definitely should not be considered equivalent. Hence, result equivalence is *not* used to define equivalence of schedules.

The safest and most general approach to defining schedule equivalence is not to make any assumption about the types of operations included in the transactions. For two schedules to be equivalent, the operations applied to each data item affected by the schedules should be applied to that item in both schedules *in the same order*. Two definitions of equivalence of schedules are generally accepted: *conflict equivalence* and *view equivalence*. We discuss each of these next.

Two schedules are said to be **conflict equivalent** if the order of any two *conflicting* operations is the same in both schedules. Recall from Section 17.4.1 that two operations in a schedule are said to *conflict* if they belong to different transactions, if they access the same database item, and if one of the two operations is a write_item operation. If

S_1	S_2
read_item(X);	read_item(X);
$X := X + 10$;	$X := X * 1.1$;
write_item(X);	write_item(X);

Figure 17.6 Two schedules that are equivalent for the initial value of $X = 100$ but are not equivalent in general.

two conflicting operations are applied in *different orders* in two schedules, the effect of the schedules can be different on either the transactions or the database, and hence the schedules are not conflict equivalent. For example, if a read and write operation occur in the order $r_1(X)$, $w_2(X)$ in a schedule S_a , and in the reverse order $w_2(X)$, $r_1(X)$ in a schedule S_b , the value read by the $r_1(X)$ operation can be different in the two schedules. Similarly, if two write operations occur in the order $w_1(X)$, $w_2(X)$ in a schedule S_a , and in the reverse order $w_2(X)$, $w_1(X)$ in another schedule S_b , the next $r(X)$ operation in the two schedules will read potentially different values; or if these are the last operations accessing item X in the schedules, the final value of item X will be different for the two schedules.

Using the notion of conflict equivalence, we define a schedule S to be **conflict serializable** if it is (conflict) equivalent to some serial schedule S' . In such a case, we can reorder the *nonconflicting* operations in S until we form the equivalent serial schedule S' . According to this definition, schedule D of Figure 17.5(c) is equivalent to the serial schedule A of Figure 17.5(a). In both these schedules, the $\text{read_item}(X)$ of T_2 reads the value of X written by T_1 , while the other read_item operations read the database values from the initial database state. In addition, T_1 is the last transaction to write item Y , and T_2 is the last transaction to write X in both schedules. Because A is a serial schedule and schedule D is equivalent to A , D is a *serializable schedule*. Notice that the operations $r_1(Y)$ and $w_1(Y)$ of schedule D do not conflict with the operations $r_2(X)$ and $w_2(X)$, since they access different data items. Hence, we can move $r_1(Y)$, $w_1(Y)$ before $r_2(X)$, $w_2(X)$, leading to the equivalent serial schedule T_1, T_2 .

Schedule C of Figure 17.5(c) is not equivalent to either of the two possible serial schedules A and B . It is not equivalent to A , because the $\text{read_item}(X)$ operation in T_2 reads the value written by T_1 in schedule A but reads the original value of X in schedule C , thus violating rule 1 for equivalence. At the same time, schedule C is not equivalent to B . Hence, schedule C is *not serializable*. If we try to reorder the operations of schedule C to find an equivalent serial schedule, we fail because $r_2(X)$ and $w_1(X)$ conflict, so we cannot move $r_1(X)$ down to get the equivalent serial schedule T_1, T_2 . Similarly, because $w_1(X)$ and $w_2(X)$ conflict, we cannot move $w_1(X)$ down to get the equivalent serial schedule T_2, T_1 .

Another, more complex definition of equivalence—called *view equivalence*, which leads to the concept of *view serializability*—is discussed in Section 17.5.4.

17.5.2 Testing for Conflict Serializability of a Schedule

There is a simple algorithm for determining the conflict serializability of a schedule. Most concurrency control methods *do not* actually test for serializability. Rather, protocols, or rules, are developed that guarantee that a schedule will be serializable. We discuss the algorithm for testing conflict serializability of schedules here to gain a better understanding of these protocols, which are discussed in Chapter 18.

Algorithm 17.1 can be used to test a schedule for conflict serializability. The algorithm looks at only the read_item and write_item operations in the schedule to construct a **precedence graph** (or **serialization graph**). A precedence graph is a **directed graph** $G = (N, E)$ that consists of a set of nodes $N = \{T_1, T_2, \dots, T_n\}$ and a set of directed edges $E = \{e_1, e_2, \dots, e_m\}$. There is one node in the graph for each transaction T_i in the

schedule. Each edge e_i in the graph is of the form $(T_j \rightarrow T_k)$, $1 \leq j \leq n$, $1 \leq k \leq n$, where T_j is called the **starting node** of e_i and T_k is called the **ending node** of e_i such that one of the operations in T_j appears in the schedule before some *conflicting operation* in T_k .

ALGORITHM 17.1 Testing conflict serializability of a schedule S

- (1) for each transaction T_i participating in schedule S
create a node labeled T_i in the precedence graph;
- (2) for each case in S where T_j executes a `read_item(X)` after a `write_item(X)` command executed by T_i
create an edge $(T_i \rightarrow T_j)$ in the precedence graph;
- (3) for each case in S where T_j executes `write_item(X)` after T_i executes a `read_item(X)`
create an edge $(T_i \rightarrow T_j)$ in the precedence graph;
- (4) for each case in S where T_j executes a `write_item(X)` command after T_i executes a `write_item(X)` command
create an edge $(T_i \rightarrow T_j)$ in the precedence graph;
- (5) the schedule S is serializable if and only if the precedence graph has no cycles;

The precedence graph is constructed as shown in Algorithm 17.1. If there is a cycle in the precedence graph, schedule S is not (conflict) serializable; if there is no cycle, S is serializable. A **cycle** in a directed graph is a *sequence* of edges $C = ((T_j \rightarrow T_k), (T_k \rightarrow T_p), \dots, (T_i \rightarrow T_j))$ with the property that the starting node of each edge—except the first edge—is the same as the ending node of the previous edge, and the starting node of the first edge is the same as the ending node of the last edge. Hence, the sequence starts and ends at the same node.

In the precedence graph, an edge from T_i to T_j means that transaction T_i must come before transaction T_j in any serial schedule that is equivalent to S , because two conflicting operations appear in the schedule in that order. If there is no cycle in the precedence graph, we can create an **equivalent serial schedule** S' that is equivalent to S , by ordering the transactions that participate in S as follows: Whenever an edge exists in the precedence graph from T_i to T_j , T_i must appear before T_j in the equivalent serial schedule S' . The latter process is known as topological sorting. Notice that the edges $(T_i \rightarrow T_j)$ in a precedence graph can optionally be labeled by the name(s) of the data item(s) that led to creating the edge.

In general, several serial schedules can be equivalent to S if the precedence graph has no cycle. On the other hand, if the precedence graph has a cycle, it is easy to show that we cannot create any equivalent serial schedule, so schedule S is not serializable. The precedence graphs created for schedules A to D, respectively, of Figure 17.5 appear in Figure 17.7(a) to (d). The precedence graph for schedule C has a cycle, so it is not serializable. The graph for schedule D has no cycle, so it is serializable; and the equivalent serial schedule is T_1 followed by T_2 . The graphs for schedules A and B have no cycles, as expected, because the schedules are *serial* and hence serializable.

Another example, in which three transactions participate, is shown in Figure 17.8. Figure 17.8(a) shows the `read_item` and `write_item` operations in each transaction. Two

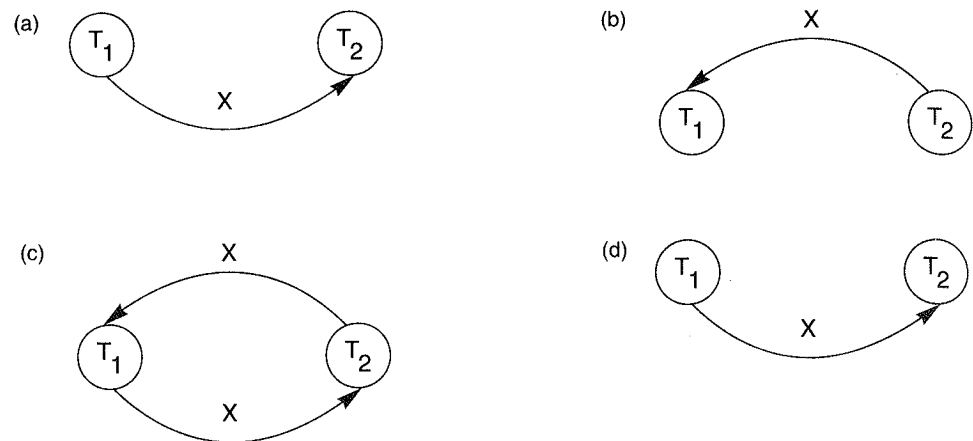


Figure 17.7 Constructing the precedence graphs for schedules A to D to test for conflict serializability. (a) Precedence graph for schedule A of Figure 17.5. (b) Precedence graph for schedule B of Figure 17.5. (c) Precedence graph for schedule C of Figure 17.5 (not serializable). (d) Precedence graph for schedule D of Figure 17.5 (serializable, equivalent to schedule A).

schedules E and F for these transactions are shown in Figure 17.8(b) and (c), respectively. We show the precedence graphs for schedules E and F in Figure 17.8(d) and (e). Schedule E is not serializable, because the corresponding precedence graph has cycles. Schedule F is serializable, and the serial schedule equivalent to F is shown in Figure 17.8(e). Although only one equivalent serial schedule exists for schedule F, in general there may be more than one equivalent serial schedule for a serializable schedule. Figure 17.8(f) shows a precedence graph representing a schedule that has two equivalent serial schedules.

17.5.3 Uses of Serializability

As we discussed earlier, saying that a schedule S is (conflict) serializable—that is, that S is (conflict) equivalent to a serial schedule—is tantamount to saying that S is correct. Being *serializable* is distinct from being *serial*, however. A serial schedule represents inefficient processing because no interleaving of operations from different transactions is permitted. This can lead to low CPU utilization while a transaction waits for disk I/O, thus slowing down processing considerably. A serializable schedule gives us the benefits of concurrent execution without giving up any correctness.

In practice, it is quite difficult to test for the serializability of a schedule. The interleaving of operations from concurrent transactions is typically determined by the operating system scheduler. Factors such as system load, time of transaction submission, and priorities of transactions contribute to the ordering of operations in a schedule by the operating system. Hence, it is practically impossible to determine how the operations of a schedule will be interleaved beforehand to ensure serializability.

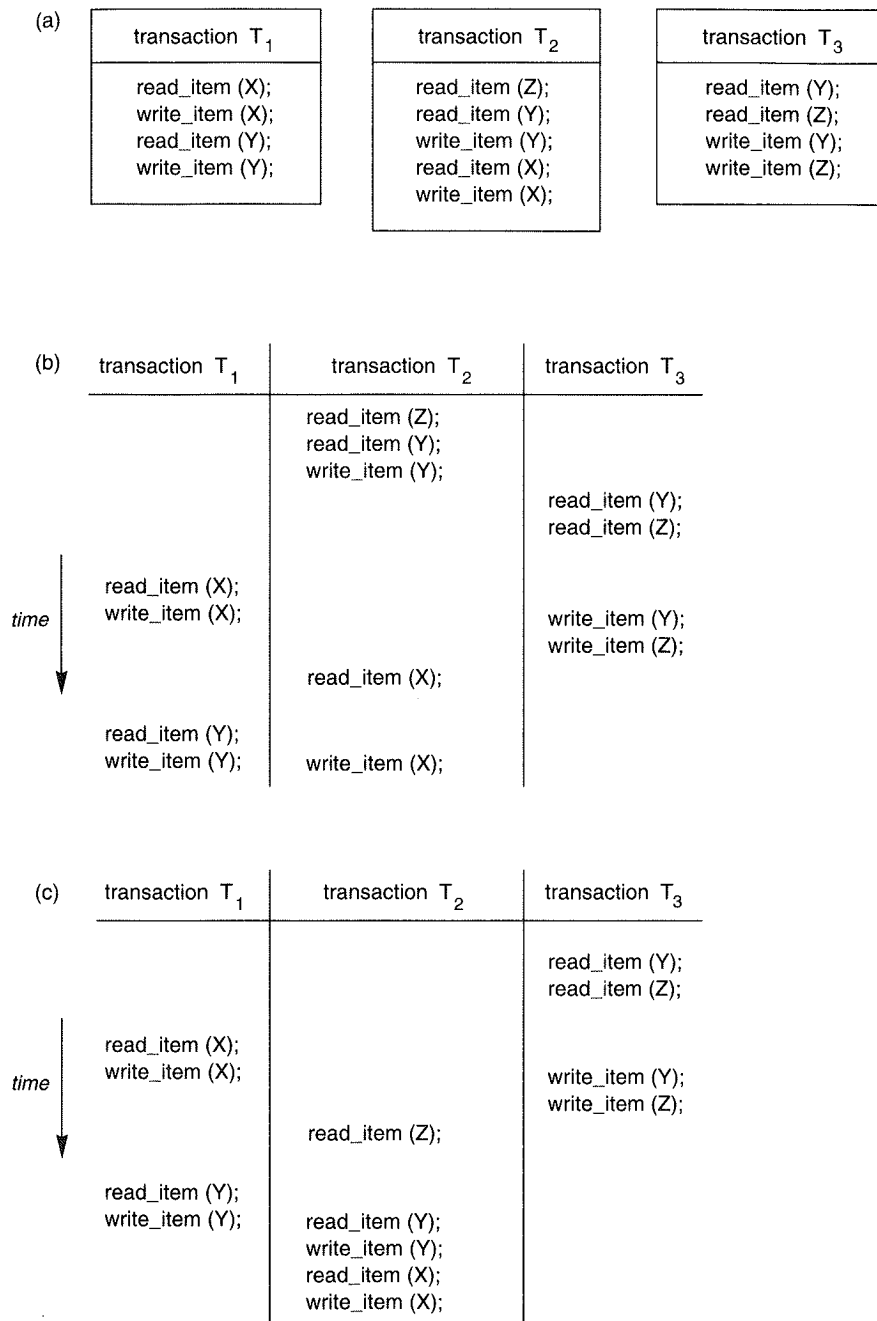


Figure 17.8 Another example of serializability testing. (a) The READ and WRITE operations of three transactions. (b) Schedule E of the transactions T_1 , T_2 , and T_3 . (c) Schedule F of the transactions T_1 , T_2 , and T_3 . (continued on next page)

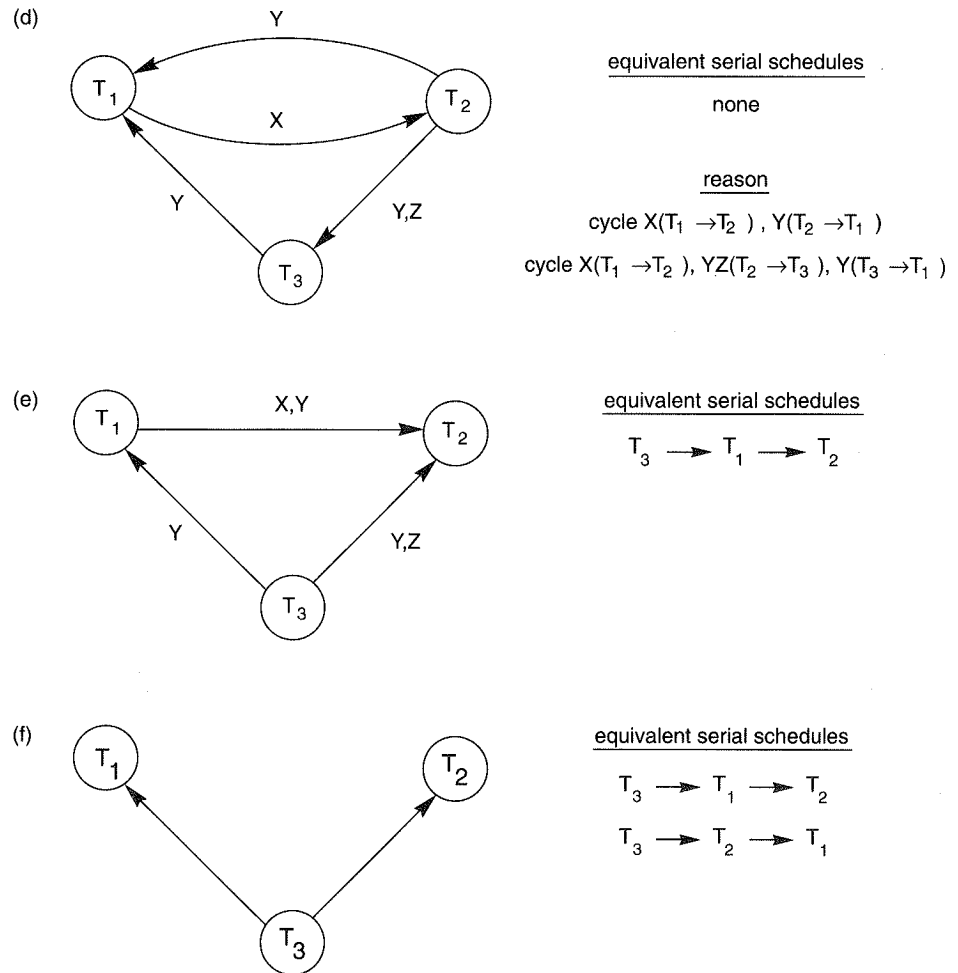


Figure 17.8 (continued) (d) Precedence graph for schedule E. (e) Precedence graph for schedule F. (f) A precedence graph with two equivalent serial schedules.

If transactions are executed at will and then the resulting schedule is tested for serializability, we must cancel the effect of the schedule if it turns out not to be serializable. This is a serious problem that makes this approach impractical. Hence, the approach taken in most practical systems is to determine methods that ensure serializability without having to test the schedules themselves for serializability after they are executed. One such method uses the theory of serializability to determine **protocols** or sets of rules that, if followed by *every* individual transaction or if enforced by a DBMS concurrency control subsystem, will ensure serializability of *all schedules in which the transactions participate*. Hence, in this approach we never have to concern ourselves with the schedules themselves.

Another problem is that, when transactions are submitted continuously to the system, it is difficult to determine when a schedule begins and when it ends. Serializability theory can be adapted to deal with this problem by considering only the committed projection of a schedule S . Recall from Section 17.4.1 that the *committed projection* $C(S)$ of a schedule S includes only the operations in S that belong to committed transactions. We can define a schedule S to be serializable if its committed projection $C(S)$ is equivalent to some serial schedule, since only committed transactions are guaranteed by the DBMS.

In Chapter 18, we discuss a number of different concurrency control protocols that guarantee serializability. The most common technique, called *two-phase locking*, is based on locking data items to prevent concurrent transactions from interfering with one another. Another technique is based on *timestamp ordering*, where each transaction is assigned a unique timestamp and the protocol ensures that any conflicting operations are executed in the order of the transaction timestamps. Other techniques are based on maintaining *multiple versions* of data items and on *certification* or *validation*.

Another factor that affects concurrency control is the **granularity** of the items—that is, what portion of the database an item represents. An item can be as small as a single value or as large as the database itself. Of course, in the latter case, very little concurrency is allowed. We discuss granularity of items in Section 18.5.

17.5.4 View Equivalence and View Serializability★

In Section 17.5.1, we defined the concepts of conflict equivalence of schedules, and of conflict serializability. Another, less restrictive definition of equivalence of schedules is called *view equivalence*. This leads to another definition of serializability called *view serializability*. Two schedules are said to be **view equivalent** if the following three conditions hold:

1. The same set of transactions participates in S and S' , and S and S' include the same operations of those transactions.
2. For any operation $r_i(X)$ of T_i in S , if the value of X read by the operation has been written by an operation $w_j(X)$ of T_j (or if it is the original value of X before the schedule started), the same condition must hold for the value of X read by operation $r_i(X)$ of T_i in S' .
3. If the operation $w_k(Y)$ of T_k is the last operation to write item Y in S , then $w_k(Y)$ of T_k must also be the last operation to write item Y in S' .

The idea behind view equivalence is that, as long as each read operation of a transaction reads the result of *the same write operation* in both schedules, the write operations of each transaction must produce the same results. The read operations are hence said to *see the same view* in both schedules. Condition 3 ensures that the final write operation on each data item is the same in both schedules, so the database state should be the same at the end of both schedules. A schedule S is said to be **view serializable** if it is view equivalent to a serial schedule.

The definitions of conflict serializability and view serializability are similar if a condition known as the **constrained write** assumption holds on all transactions in the sched-

ule. This condition states that any write operation $w_i(X)$ in T_i is preceded by a $r_i(X)$ in T_i and that the value written by $w_i(X)$ in T_i depends only on the value of X read by $r_i(X)$. This assumes that computation of the new value of X is a function $f(X)$ based on the old value of X read from the database. However, the definition of view serializability is less restrictive than that of conflict serializability under the **unconstrained write** assumption, where the value written by an operation $w_i(X)$ in T_i can be independent of any old value from the database. This is called a **blind write**, and is illustrated by the following schedule of three transactions T_1 : $r_1(X)$, $w_1(X)$; T_2 : $w_2(X)$; and T_3 : $w_3(X)$:

S_a : $r_1(X)$; $w_2(X)$; $w_1(X)$; $w_3(X)$; c_1 ; c_2 ; c_3 ;

In S_a , the operations $w_2(X)$ and $w_3(X)$ are blind writes, since T_1 and T_3 do not read the value of X . The schedule S_a is view serializable, since it is view equivalent to the serial schedule T_1, T_2, T_3 . However, S_a is not conflict serializable, since it is not conflict equivalent to any serial schedule.

It has been shown that any conflict-serializable schedule is also view serializable, but not vice versa, as illustrated by the preceding example. There is an algorithm to test whether a schedule S is view serializable or not. However, the problem of testing for view serializability has been shown to be NP-complete, meaning that finding an efficient algorithm for this problem is highly unlikely.

17.5.5 Other Types of Equivalence of Schedules*

Serializability of schedules is sometimes considered to be too restrictive as a condition for ensuring the correctness of concurrent executions. Some applications can produce schedules that are correct by satisfying conditions less stringent than either conflict serializability or view serializability. An example is the type of transactions known as debit-credit transactions—for example, those that apply deposits and withdrawals to a data item whose value is the current balance of a bank account. The semantics of debit-credit operations is that they update the value of a data item X by either subtracting from or adding to the value of the data item. Because addition and subtraction operations are commutative—that is, they can be applied in any order—it is possible to produce correct schedules that are not serializable. For example, consider the following two transactions, each of which may be used to transfer an amount of money between two bank accounts:

T_1 : $r_1(X)$; $X := X - 10$; $w_1(X)$; $r_1(Y)$; $Y := Y + 10$; $w_1(Y)$;
 T_2 : $r_2(Y)$; $Y := Y - 20$; $w_2(Y)$; $r_2(Z)$; $Z := Z + 20$; $w_2(Z)$;

Consider the following schedule S for the two transactions:

S : $r_1(X)$; $w_1(X)$; $r_2(Y)$; $w_2(Y)$; $r_1(Y)$; $w_1(Y)$; $r_2(Z)$; $w_2(Z)$;

With the additional knowledge, or *semantics*, that the operations between each $r_i(X)$ and $w_i(X)$ are commutative, we know that the order of executing the sequences consisting of (read, update, write) is not important as long as each (read, update, write) sequence is not interrupted. Hence, the schedule S is considered to be correct even though it is not serializable. Recently, researchers have been working on extending concurrency control theory to deal with cases where serializability is considered to be too restrictive as a condition for correctness of schedules.

17.6 Summary

In this chapter we discussed DBMS concepts for transaction processing. In Section 17.1 we introduced the concept of a database transaction and the operations relevant to transaction processing. We compared single-user systems to multiuser systems and then presented examples of how uncontrolled execution of concurrent transactions in a multiuser system can lead to incorrect results and database values. We also discussed the various types of failures that may occur during transaction execution.

In Section 17.2, we first introduced the typical states that a transaction passes through during execution; then we discussed several concepts that are used in recovery and concurrency control methods. The system log keeps track of database accesses and uses this information to recover from failures. A transaction either succeeds and reaches its commit point or fails and has to be rolled back. A committed transaction has its changes permanently recorded in the database. Checkpoints are used to indicate that all committed transactions up to the checkpoint have recorded their updates permanently in the database. In Section 17.3, we presented an overview of the desirable properties of transactions—namely, atomicity, consistency preservation, isolation, and durability—which are often referred to as the ACID properties.

In Section 17.4, we first defined a schedule (or history) as an execution sequence of the operations of several transactions with possible interleaving. We then characterized schedules in terms of their recoverability. Recoverable schedules ensure that, once a transaction commits, it never needs to be undone. Cascadeless schedules ensure that no aborted transaction requires the cascading abort of other transactions. Strict schedules allow a simple recovery scheme consisting of rewriting the old values of items that have been changed by an aborted transaction.

In Section 17.5 we defined equivalence of schedules and saw that a serializable schedule is equivalent to some serial schedule. We defined the concepts of conflict equivalence and view equivalence, leading to definitions for conflict serializability and view serializability. A serializable schedule is considered correct. We then presented algorithms for testing the (conflict) serializability of a schedule. Finally, we discussed why testing for serializability is impractical in a real system, although it can be used to define and verify concurrency control protocols, and we briefly mentioned less restrictive definitions of schedule equivalence.

We will discuss a number of concurrency control techniques in Chapter 18, and we will discuss the techniques for recovery from transaction failures in Chapter 19.

Review Questions

- 17.1. What is meant by *concurrent execution of database transactions* in a multiuser system? Discuss why concurrency control is needed, and give informal examples.
- 17.2. Discuss the different types of transaction failures. What is meant by *catastrophic failure*?
- 17.3. Discuss the actions taken by the `read_item` and `write_item` operations on a database.

- 17.4. Draw a state diagram, and discuss the typical states that a transaction goes through during execution.
- 17.5. What is the system log used for? What are the typical kinds of entries in a system log? What are checkpoints, and why are they important? What are transaction commit points, and why are they important?
- 17.6. Discuss the atomicity, durability, isolation, and consistency preservation properties of a database transaction.
- 17.7. What is a schedule (history)? Define the concepts of recoverable, cascadeless, and strict schedules, and compare them in terms of their recoverability.
- 17.8. Discuss the different measures of transaction equivalence. What is the difference between conflict equivalence and view equivalence?
- 17.9. What is a serial schedule? What is a serializable schedule? Why is a serial schedule considered correct? Why is a serializable schedule considered correct?
- 17.10. What is the difference between the constrained write and the unconstrained write assumptions? Which is more realistic?
- 17.11. Discuss how serializability is used to enforce concurrency control in a database system. Why is serializability sometimes considered too restrictive as a measure of correctness for schedules?

Exercises

- 17.12. Change transaction T_2 in Figure 17.2b to read:

```

read_item(x);
X := X + M;
if X > 90 then exit
    else write_item(X);

```

Discuss the final result of the different schedules in Figure 17.3, where $M = 2$ and $N = 2$, with respect to the following questions. Does adding the above condition change the final outcome? Does the outcome obey the implied consistency rule (that the capacity of X is 90)?

- 17.13. Repeat Exercise 17.12, adding a check in T_1 so that Y does not exceed 90.
- 17.14. Add the operation commit at the end of each of the transactions T_1 and T_2 from Figure 17.2; then list all possible schedules for the modified transactions. Determine which of the schedules are recoverable, which are cascadeless, and which are strict.
- 17.15. List all possible schedules for transactions T_1 and T_2 from Figure 17.2, and determine which are conflict serializable (correct) and which are not.
- 17.16. How many *serial* schedules exist for the three transactions in Figure 17.8(a)?
- 17.17. Write a program to create all possible schedules for the three transactions in Figure 17.8(a), and to determine which of those schedules are conflict serializable and which are not. For each conflict serializable schedule, your program should print the schedule and list all equivalent serial schedules.

Selected Bibliography

The concept of atomic transaction is discussed in Gray (1981). The book by Bernstein, Hadzilacos, and Goodman (1987) is devoted to concurrency control and recovery techniques in both centralized and distributed database systems; it is an excellent reference. The book by Papadimitriou (1986) offers a more theoretical perspective. A large reference book of over 1000 pages by Gray and Reuter (1993) offers a more practical perspective of transaction processing concepts and techniques. Two edited volumes—Elmagarmid (1992) and Bhargava (1989)—offer collections of research papers on transaction processing.

The concepts of serializability are introduced in Gray et al. (1975). View serializability is defined in Yannakakis (1984). Recoverability of schedules is discussed in Hadzilacos (1983, 1986).

Recovery Techniques

In this chapter we discuss some of the techniques that can be used to recover from transaction failures. We have already discussed the different causes of failure (such as system crashes and transaction errors), in Section 17.1.4. We have also covered many of the concepts that are used by recovery processes (such as the system log, checkpoints, and commit points), in Section 17.2.

We first introduce recovery concepts in Section 19.1, including an outline of recovery procedures and categorization of recovery algorithms. We also discuss write-ahead logging, in-place versus shadow updates, and the process of rolling back (undoing) the effect of a transaction. In Section 19.2, we present recovery techniques based on deferred update, also known as the NO-UNDO/REDO technique. In Section 19.3, we discuss recovery techniques based on immediate update; these include the UNDO/REDO and UNDO/NO-REDO algorithms. We discuss the technique known as shadowing or shadow paging, which can be categorized as a NO-UNDO/NO-REDO algorithm in Section 19.4. Recovery in multidatabase transactions is briefly discussed in Section 19.5. Finally, techniques for recovery from catastrophic failure are discussed in Section 19.6.

The techniques we discuss here are *not* descriptions of recovery as implemented in a specific system. Our emphasis is on conceptually describing several different approaches to recovery. For descriptions of recovery features in specific systems, the reader should consult the bibliographic notes and the user manuals for those systems.

Recovery techniques are often intertwined with the concurrency control mechanisms. Certain recovery techniques are best used with specific concurrency control meth-

ods. We will attempt to discuss recovery concepts independently of concurrency control mechanisms, but we will discuss the circumstances under which a particular recovery mechanism is best used with a certain concurrency control protocol.

19.1 Recovery Concepts

19.1.1 Recovery Outline

Recovery from transaction failures usually means that the database is *restored* to some state from the past so that a correct state—close to the time of failure—can be *reconstructed* from that past state. To do this, the system must keep information about changes to data items during transaction execution outside the database. This information is typically kept in the **system log**, as discussed in Section 17.2.2. A typical strategy for recovery may be summarized informally as follows:

1. If there is extensive damage to a wide portion of the database due to catastrophic failure, such as a disk crash, the recovery method restores a past copy of the database that was *dumped* to archival storage (typically tape) and reconstructs a more current state by reapplying or *redoing* committed transaction operations from the log up to the time of failure.
2. When the database is not physically damaged but has become inconsistent due to noncatastrophic failures of types 1 through 4 of Section 17.1.4, the strategy is to reverse the changes that caused the inconsistency by *undoing* some operations. It may also be necessary to *redo* some operations in order to restore a consistent state of the database, as we shall see. In this case we do not need a complete archival copy of the database. Rather, the entries kept in the system log are consulted during recovery.

We can distinguish two main techniques for recovery from noncatastrophic transaction failures. The **deferred update** techniques do not actually update the database until *after* a transaction reaches its commit point; then the updates are recorded in the database. Before commit, all transaction updates are recorded in the local transaction workspace. During commit, the updates are first recorded persistently in the log and then written to the database. If a transaction fails before reaching its commit point, it will not have changed the database in any way, so UNDO is not needed. It may be necessary to REDO the effect of the operations of a committed transaction from the log, because their effect may not yet have been recorded in the database. Hence, deferred update is also known as the **NO-UNDO/REDO algorithm**. We discuss this technique in Section 19.2.

In the **immediate update** techniques, the database may be updated by some operations of a transaction *before* the transaction reaches its commit point. However, these operations are typically recorded in the log *on disk* by force-writing before they are applied to the database, making recovery still possible. If a transaction fails after recording some changes in the database but before reaching its commit point, the effect of its operations on the database must be undone; that is, the transaction must be rolled back. In the general case of immediate update, both *undo* and *redo* are required during recovery,

so it is known as the **UNDO/REDO algorithm**. A variation of the algorithm where all updates are recorded in the database before a transaction commits requires *undo* only, so it is known as the **UNDO/NO-REDO algorithm**. We discuss these techniques in Section 19.3.

19.1.2 System Concepts for Recovery

The recovery process is often closely intertwined with operating system functions—in particular, the buffering and caching of disk pages in main memory. Typically, one or more disk pages that include the data item to be updated are **cached** into a main memory buffer and then updated in memory before being written back to disk. The caching of disk pages is traditionally an operating system function, but because of its importance to the efficiency of recovery procedures, it is sometimes handled by the DBMS by calling low-level operating systems routines. In general, it is convenient to consider that each data item corresponds to one disk page for recovery purposes.

Typically a collection of in-memory buffers, called the **DBMS cache**, is kept under the control of the DBMS for the purpose of holding database items. A **directory** for the cache is used to keep track of which database items are in the buffers. This can be a table of <item name, buffer location> entries. When the DBMS requests action on some item, it first checks the directory to determine whether the item is in the cache. If it is not in the cache, the item must be located on disk, and the appropriate disk pages are copied into the cache. It may be necessary to **flush** some of the cache buffers to make space available for the new item. Some page-replacement strategy from operating systems, such as least recently used (LRU) or first-in-first-out (FIFO), can be used to select the buffers for flushing.

Associated with each item in the cache is a **dirty bit**, which can be included in the directory entry, to indicate whether or not the item has been modified. When the item is first read from disk into a cache buffer, the buffer directory is updated with the new item name, and the dirty bit is set to 0 (zero). As soon as the item is modified, the dirty bit for the corresponding directory entry is set to 1 (one). When an item is flushed, it is written back to disk only if its dirty bit is 1.

Two main strategies are employed when flushing a modified data item back to disk. The first strategy, known as **in-place updating**, writes the data item in the *same disk location*, thus overwriting the old value of the data item on disk. Hence, a single copy of each data item is maintained on disk. The second strategy, known as **shadowing**, writes a new item at a different disk location, so multiple copies of a data item can be maintained. In general, the old value of the data item before updating is called the **before image** (BFIM), and the new value after updating is called the **after image** (AFIM). In shadowing, both the BFIM and the AFIM are kept on disk; hence, it is not strictly necessary to maintain a log for recovering. We discuss recovery based on shadowing in Section 19.4.

When in-place updating is used, it is necessary to use a log for recovery (see Section 17.2.2). In this case, the recovery mechanism must ensure that the BFIM of the data item is recorded in the appropriate log entry and that log entry is flushed to disk before the BFIM is overwritten with the AFIM. This process is generally known as a **write-ahead logging**. Before we can describe a protocol for write-ahead logging, we need to distinguish

between two types of log entries: those needed for UNDO, and those needed for REDO. A **REDO-type log entry** is a log entry corresponding to a `write_item` operation of some transaction that includes the new value (AFIM) of the item; such an entry is needed if the recovery technique must *redo* the effect of the operation from the log by setting the item value in the database to its AFIM. The **UNDO-type log entries** include `write_item` entries that include the old value (BFIM) of the item; such an entry is needed if the recovery technique must *undo* the effect of the operation from the log by setting the item value in the database back to its BFIM. In addition, when cascading rollback is possible, `read_item` entries in the log are considered to be UNDO-type entries (see Section 19.1.3).

To permit recovery when in-place updating is used, the appropriate entries required for recovery must be permanently recorded in the log on disk before changes are applied to the database. For example, consider the following **write-ahead logging (WAL)** protocol for a recovery algorithm that requires both UNDO and REDO:

1. The before image of an item cannot be overwritten by its after image on disk until all UNDO-type log records for the updating transaction—up to this point in time—have been force-written to disk.
2. The commit operation of a transaction cannot be completed until all the REDO-type and UNDO-type log records for that transaction have been force-written to disk.

To facilitate the recovery process, the DBMS recovery subsystem maintains a number of lists related to the transactions being processed in the system. These include a list for **active transactions** that have started but not committed as yet, a list of all **committed transactions** since the last checkpoint, and a list of **aborted transactions** since the last checkpoint. Maintaining these lists makes the recovery process more efficient.

19.1.3 Transaction Rollback

If a transaction fails for whatever reason after updating the database, it may be necessary to **roll back** or UNDO the transaction. Any data item values that have been changed by the transaction must be returned to their previous values (BFIMs). The log entries are used to recover the old values of data items that must be rolled back.

If a transaction T is rolled back, any transaction S that has, in the interim, read the value of some data item X written by T must also be rolled back. Similarly, once S is rolled back, any transaction R that has read the value of some data item Y written by S must also be rolled back; and so on. This phenomenon is called **cascading rollback**, and can occur when the recovery protocol ensures *recoverable* schedules but does not ensure *strict* or *cascadeless* schedules (see Section 17.4.2). Cascading rollback, understandably, can be quite time-consuming. That is why most recovery mechanisms are designed such that cascading rollback is *never required*.

Figure 19.1 shows an example where cascading rollback is required. The read and write operations of three individual transactions are shown in Figure 19.1(a). Figure 19.1(b) shows the system log at the point of a system crash for a particular execution schedule of these transactions. The values of data items A, B, C, and D, which are used

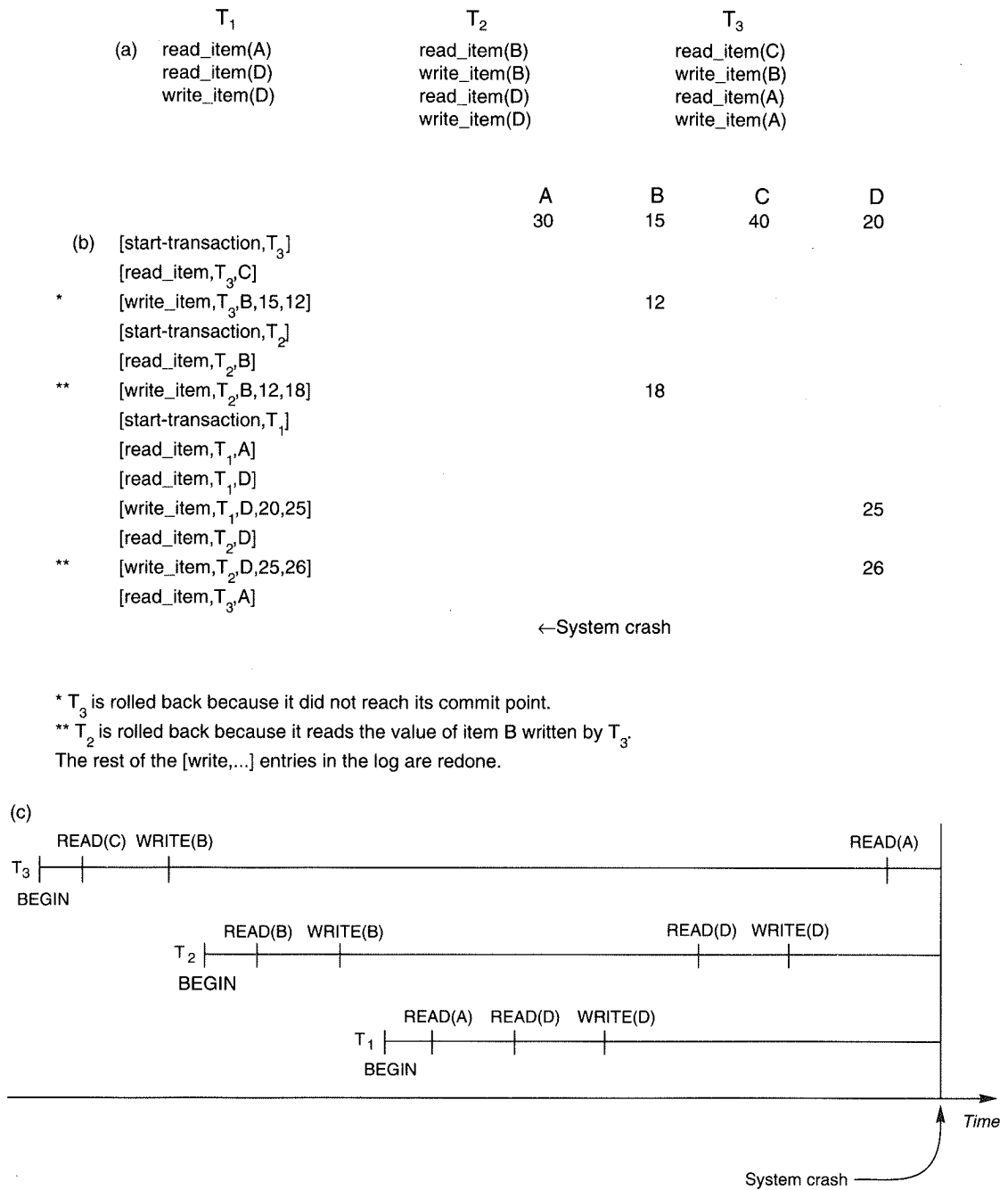


Figure 19.1 Cascading rollback. (a) The read and write operations of three transactions. (b) System log at point of crash. (c) Operations before the crash.

by the transactions, are shown to the right of the system log entries. We assume that the original item values, shown in the first line, are $A = 30$, $B = 15$, $C = 40$, and $D = 20$. At the point of system failure, transaction T_3 has not reached its conclusion and must be rolled back. The WRITE operations of T_3 , marked by a single * in Figure 19.1(b), are the T_3 operations that are undone during transaction rollback. Figure 19.1(c) graphically shows the operations of the different transactions along the time axis.

We must now check for cascading rollback. From Figure 19.1(c) we see that transaction T_2 reads the value of item B that was written by transaction T_3 ; this can also be determined by examining the log. Because T_3 is rolled back, T_2 must now be rolled back, too. The WRITE operations of T_2 , marked by ** in the log, are the ones that are undone. Note that only write_item operations need to be undone during transaction rollback; read_item operations are recorded in the log only to determine whether cascading rollback of additional transactions is necessary.

If rollback of transactions is *never* required by the recovery method (because the method guarantees cascadeless or strict schedules), we can keep *more limited information* in the system log. There is also no need to record any read_item operations in the log, because these are needed only for determining cascading rollback.

19.2 Recovery Techniques Based on Deferred Update

The idea behind deferred update techniques is to defer or postpone any actual updates to the database itself until the transaction completes its execution successfully and reaches its commit point. During transaction execution, the updates are recorded only in the log and in the transaction workspace. After the transaction reaches its commit point and the log is force-written to disk, the updates are recorded in the database itself. If a transaction fails before reaching its commit point, there is no need to undo any operations, because the transaction has not affected the database in any way.

We can state a typical deferred update protocol as follows:

1. A transaction cannot change the database until it reaches its commit point.
2. A transaction does not reach its commit point until all its update operations are recorded in the log *and* the log is force-written to disk.

Notice that step 2 of this protocol is a variation on the write-ahead logging (WAL) protocol. Because the database is never updated until after the transaction commits, there is never a need to UNDO any operations. Hence, this technique is known as the NO-UNDO/REDO algorithm. The REDO is needed in case the system fails after the transaction commits but before all its changes are recorded in the database. In this case, the transaction operations are redone from the log entries.

Usually, the method of recovery from failure is closely related to the concurrency control method in multiuser systems. First we discuss recovery in single-user systems, where no concurrency control is needed, so that we can understand the recovery process

independently of any concurrency control method. Then we discuss how concurrency control may affect the recovery process.

19.2.1 Recovery Using Deferred Update in a Single-user Environment

In such an environment, the recovery algorithm can be rather simple. The algorithm RDU_S (recovery using deferred update in a single-user environment) uses a REDO procedure, given subsequently, for redoing certain write_item operations; it works as follows:

PROCEDURE RDU_S Use two lists of transactions: the committed transactions since the last checkpoint, and the active transactions (at most one transaction will fall in this category, because the system is single-user). Apply the REDO operation to all the write_item operations of the committed transactions from the log *in the order in which they were written to the log*. Restart the active transactions.

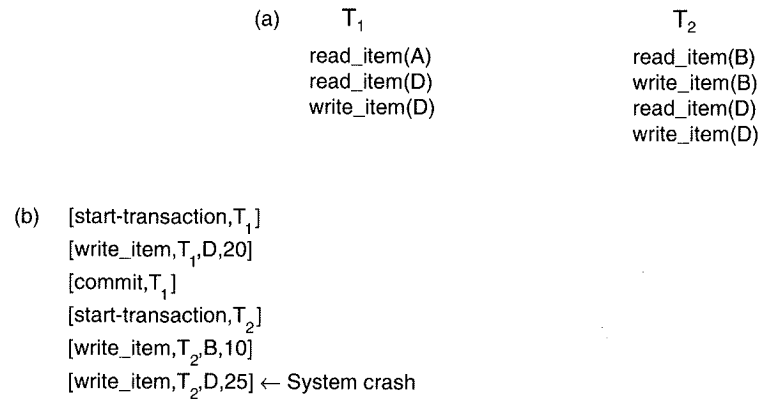
The REDO procedure is defined as follows:

REDO(WRITE_OP) Redoing a write_item operation WRITE_OP consists of examining its log entry [write_item,T,X,new_value] and setting the value of item X in the database to new_value, which is the after image (AFIM).

The REDO operation is required to be **idempotent**; that is, executing it over and over is equivalent to executing it just once. In fact, the whole recovery process should be idempotent. This is so because, if the system were to fail during the recovery process, the next recovery attempt might REDO certain write_item operations that had already been redone during the first recovery process. The result of recovery from a crash *during recovery* should be the same as the result of recovering *when there is no crash during recovery*!

Notice that the transaction in the active list will have had no effect on the database because of the deferred update protocol and is ignored completely by the recovery process. It is *implicitly rolled back*, because none of its operations were reflected in the database. However, the transaction must now be restarted, either automatically by the recovery process or manually by the user.

Figure 19.2 shows an example of recovery in a single-user environment, where the first failure occurs during execution of transaction T_2 , as shown in Figure 19.2(b). The recovery process will redo the [write_item, T_1 ,D,20] entry in the log by resetting the value of item D to 20 (its new value). The [write, T_2 , . . .] entries in the log are ignored by the recovery process because T_2 is not committed. If a second failure occurs during recovery from the first failure, the same recovery process is repeated from start to finish, with identical results.



The [write-item, ...] operations of T_1 are redone.

T_2 log entries are ignored by the recovery process.

Figure 19.2 Recovery using deferred update in a single-user environment. (a) The read and write operations of two transactions. (b) System log at the point of crash.

19.2.2 Deferred Update with Concurrent Execution in a Multiuser Environment

For multiuser systems with concurrency control, the recovery process may be more complex, depending on the protocols used for concurrency control. In many cases, the concurrency control and recovery processes are interrelated. In general, the greater the degree of concurrency we wish to achieve, the more difficult the task of recovery becomes.

Consider a system in which concurrency control uses two-phase locking and prevents deadlock by preassigning all locks to items needed by a transaction before the transaction starts execution. To combine the deferred update method for recovery with this concurrency control technique, we can keep all the locks on items in effect *until the transaction reaches its commit point*. After that, the locks can be released. This ensures strict and serializable schedules. Assuming that [checkpoint] entries are included in the log, a possible recovery algorithm for this case, which we call RDU_M (recovery using deferred update in a multiuser environment), is given next. This procedure uses the REDO procedure defined earlier.

PROCEDURE RDU_M (WITH CHECKPOINTS)

Use two lists of transactions maintained by the system: the committed transactions T since the last checkpoint, and the active transactions T' . REDO all the WRITE operations of the committed transactions from the log, *in the order in which they were written into the log*. The transactions that are active and did not commit are effectively canceled and must be resubmitted.

Figure 19.3 shows a possible schedule of executing transactions. When the checkpoint was taken at time t_1 , transaction T_1 had committed, whereas transactions T_3 and T_4 had not. Before the system crash at time t_2 , T_3 and T_2 were committed but not T_4 and T_5 . According to the RDU_M method, there is no need to redo the write_item operations of transaction T_1 —or any transactions committed before the last checkpoint time t_1 . Write_item operations of T_2 and T_3 must be redone, however, because both transactions reached their commit points after the last checkpoint. Recall that the log is force-written before committing a transaction. Transactions T_4 and T_5 are ignored: they are effectively canceled or rolled back because none of their write_item operations were recorded in the database under the deferred update protocol. We will refer to Figure 19.3 later to illustrate other recovery protocols.

We can make the NO-UNDO/REDO recovery algorithm *more efficient* by noting that, if a data item X has been updated more than once by committed transactions, it is only necessary to REDO the *last update of X* from the log during recovery, since the other updates would be overwritten by this last REDO in any case. In this case, we start from the *end of the log*; then, whenever an item is redone, it is added to a list of redone items. Before REDO is applied to an item, the list is checked; if the item appears on the list, it is not redone, since its last value has already been recovered.

A drawback of the method described here is that it limits the concurrent execution of transactions because *all items remain locked until the transaction reaches its commit point*. The method's main benefit is that transaction operations never need to be undone, for two reasons:

1. A transaction does not record its changes in the database until it reaches its commit point—that is, until it completes its execution successfully. Hence, a transaction is never rolled back because of failure during transaction execution.
2. A transaction will never read the value of an item that is written by an uncommitted transaction, because items remain locked until a transaction reaches its commit point. Hence, no cascading rollback will occur.

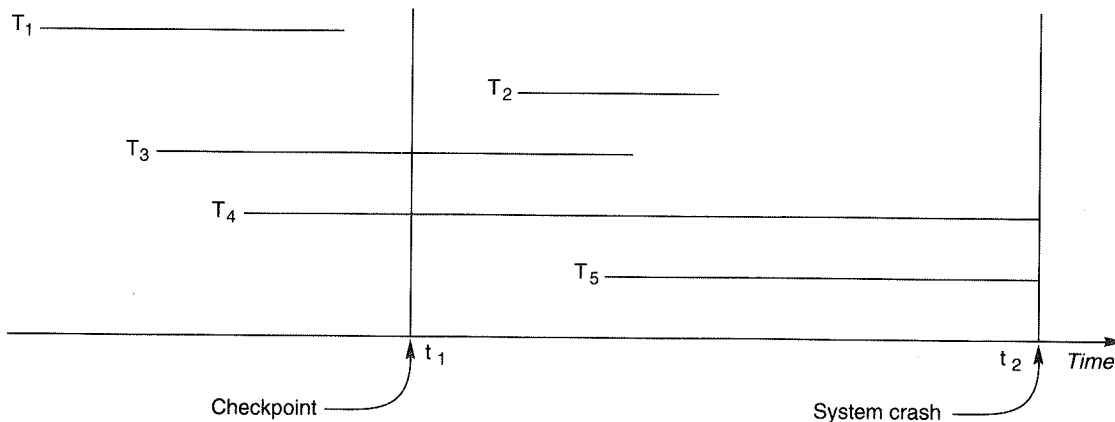


Figure 19.3 Example of recovery in a multiuser environment.

Figure 19.4 shows an example of recovery for a multiuser system that utilizes the recovery and concurrency control method just described.

19.2.3 Transaction Actions That Do Not Affect the Database

In general, a transaction will have actions that do not affect the database, such as generating and printing messages or reports from information retrieved from the database. If a transaction fails before completion, we may not want the user to get these reports, since the transaction has failed to complete. Hence, such reports should be generated *only after the transaction reaches its commit point*. A common method of dealing with such actions is to issue the commands that generate the reports but keep them as batch jobs. The batch jobs are executed only after the transaction reaches its commit point. If the transaction does not reach its commit point because of a failure, the batch jobs are canceled.

	T ₁	T ₂	T ₃	T ₄
(a)	read_item(A) read_item(D) write_item(D)	read_item(B) write_item(B) read_item(D) write_item(D)	read_item(A) write_item(A) read_item(C) write_item(C)	read_item(B) write_item(B) read_item(A) write_item(A)
(b)	[start_transaction,T ₁] [write_item,T ₁ ,D,20] [commit,T ₁] [checkpoint] [start_transaction,T ₄] [write_item,T ₄ ,B,15] [write_item,T ₄ ,A,20] [commit,T ₄] [start_transaction,T ₂] [write_item,T ₂ ,B,12] [start_transaction,T ₃] [write_item,T ₃ ,A,30] [write_item,T ₂ ,D,25] ←System crash			

T₂ and T₃ are ignored because they did not reach their commit points.

T₄ is redone because its commit point is after the last system checkpoint.

Figure 19.4 Recovery using deferred update with concurrent transactions. (a) The read and write operations of four transactions. (b) System log at the point of crash.

19.3 Recovery Techniques Based on Immediate Update★

In these techniques, when a transaction issues an update command, the database can be updated “immediately,” without any need to wait for the transaction to reach its commit point. In many of these techniques, however, an update operation must still be recorded in the log (on disk) *before* it is applied to the database so that we can recover, in case of failure, by using the write-ahead logging protocol (see Section 19.1.2).

When immediate update is allowed, provisions must be made for *undoing* the effect of update operations on the database, because a transaction can fail after it has applied some updates to the database itself. Hence, recovery schemes based on immediate update must include the capability to roll back a transaction by undoing the effect of its write_item operations.

In general, we can distinguish two main categories of immediate update algorithms. If the recovery technique ensures that all updates of a transaction are recorded in the database on disk *before the transaction commits*, there is never a need to REDO any operations of committed transactions. Such an algorithm is called **UNDO/NO-REDO**. On the other hand, if the transaction is allowed to commit before all its changes are written to the database, we have the most general recovery algorithm, known as **UNDO/REDO**. This is also the most complex technique. Next, we discuss two examples of UNDO/REDO algorithms and leave it as an exercise for the reader to develop the UNDO/NO-REDO variation.

19.3.1 UNDO/REDO Recovery Based on Immediate Update in a Single-user Environment

We first consider a single-user system so that we can examine the recovery process separately from concurrency control. If a failure occurs in a single-user system, the executing transaction at the time of failure may have recorded some changes in the database. The effect of all such operations must be undone as part of the recovery process. Hence, the recovery algorithm needs an UNDO procedure, described subsequently, to undo the effect of certain write_item operations that have been applied to the database following examination of their system log entry. The recovery algorithm RIU_S (recovery using immediate update in a single-user environment) also uses the REDO procedure defined earlier.

PROCEDURE RIU_S

1. Use two lists of transactions maintained by the system: the committed transactions since the last checkpoint, and the active transactions (at most one transaction will fall in this category, because the system is single-user).
2. Undo all the write_item operations of the *active* transaction from the log, using the UNDO procedure described hereafter.
3. Redo all the write_item operations of the *committed* transactions from the log, in the order in which they were written in the log, using the REDO procedure.

The UNDO procedure is defined as follows:

UNDO(WRITE_OP) Undoing a write_item operation WRITE_OP consists of examining its log entry [write_item,T,X,old_value,new_value] and setting the value of item X in the database to old_value which is the before image (BFIM). Undoing a number of write_item operations from one or more transactions from the log must proceed in the *reverse order* from the order in which the operations were written in the log.

19.3.2 UNDO/REDO Immediate Update with Concurrent Execution

When concurrent execution is permitted, the recovery process again depends on the protocols used for concurrency control. The procedure RIU_M (recovery using immediate updates for a multiuser environment) outlines a recovery technique for concurrent transactions with immediate update. Assume that the log includes checkpoints and that the concurrency control protocol produces *strict schedules*—as, for example, the strict two-phase locking protocol does. Recall that a strict schedule does not allow a transaction to read or write an item unless the transaction that last wrote the item has committed. However, deadlocks can occur in strict two-phase locking, thus requiring UNDO of transactions. For a strict schedule, UNDO of an operation requires changing the item back to its old value (BFIM).

PROCEDURE RIU_M

1. Use two lists of transactions maintained by the system: the committed transactions since the last checkpoint, and the active transactions.
2. Undo all the write_item operations of the *active* (uncommitted) transactions, using the UNDO procedure. The operations should be undone in the reverse of the order in which they were written into the log.
3. Redo all the write_item operations of the *committed* transactions from the log, in the order in which they were written into the log.

19.4 Shadow Paging*

This recovery scheme does not require the use of a log in a single-user environment. In a multiuser environment, it may require a log if a log is needed for the concurrency control method.* Shadow paging considers the database to be made up of a number of fixed-size disk pages (or disk blocks)—say, n —for recovery purposes. A **page table** (or **directory**) with n entries is constructed, where the i^{th} page table entry points to the i^{th} database page on disk. The page table is kept in main memory if it is not too large, and all references—reads or writes—to database pages on disk go through the page table. When a transaction begins executing, the **current page table**—whose entries point to the most recent or current database pages on disk—is copied into a **shadow page table**. The shadow page table is then saved on disk while the current page table is used by the transaction.

*For example, the experimental DBMS SYSTEM R uses shadow paging along with checkpointing and logging.

During transaction execution, the shadow page table is *never* modified. When a write_item operation is performed, a new copy of the modified database page is created, but the old copy of that page is *not overwritten*. Instead, the new page is written elsewhere—on some previously unused disk block. The current page table entry is modified to point to the new disk block, whereas the shadow page table is not modified and continues to point to the old unmodified disk block. Figure 19.5 illustrates the concepts of a shadow page table and a current page table. For pages updated by the transaction, two versions are kept. The old version is referenced by the shadow page table, and the new version by the current page table.

To recover from a failure during transaction execution, it is sufficient to free the modified database pages and to discard the current page table. The state of the database before transaction execution is available through the shadow page table, and that state is recovered by reinstating the shadow page table so that it becomes the current page table once more. The database thus is returned to its state prior to the transaction that was executing when the crash occurred, and any modified pages are discarded. Committing a transaction corresponds to discarding the previous shadow page table and freeing the old page tables that it references. Since recovery involves neither undoing nor redoing data items, this technique can be categorized as a NO-UNDO/NO-REDO technique for recovery.

The advantage of shadow paging is that it makes undoing the effect of the executing transaction very simple. There is no need to undo or redo any transaction operations. In a multiuser environment with concurrent transactions, logs and checkpoints must be incorporated into the shadow paging technique. One disadvantage of shadow paging is

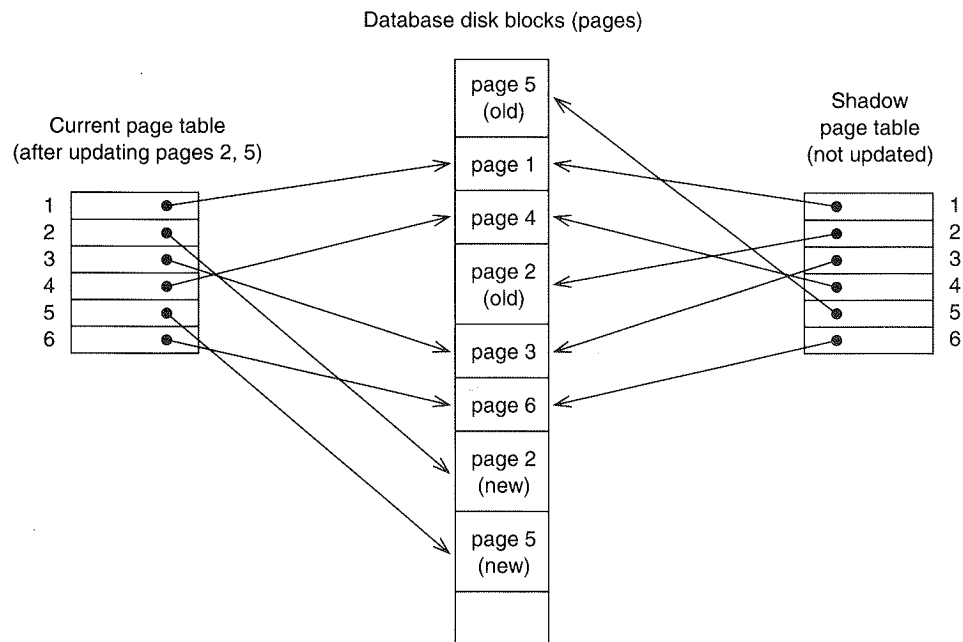


Figure 19.5 Shadow paging.

that the updated database pages change location on disk. This makes it difficult to keep related database pages close together on disk without complex storage management strategies. Furthermore, if the page table (directory) is large, the overhead of writing shadow page tables to disk as transactions commit is significant. A further complication is how to handle **garbage collection** when a transaction commits. The old pages referenced by the shadow page that have been updated must be released and added to a list of free pages for future use. These pages are no longer needed after the transaction commits, and the current page table replaces the shadow page table to become the valid page table.

19.5 Recovery in Multidatabase Transactions*

So far, we have implicitly assumed that a transaction accesses a single database. In some cases a single transaction, called a **multidatabase transaction**, may require access to multiple databases. These databases may even be stored on different types of DBMSs; for example, some DBMSs may be relational, whereas others are hierarchical or network DBMSs. In such a case, each DBMS involved in the multidatabase transaction will have its own recovery technique and transaction manager separate from those of the other DBMSs. This situation is somewhat similar to the case of a distributed database management system (see Chapter 23), where parts of the database reside at different sites that are connected by a communication network.

To maintain the atomicity of a multidatabase transaction, it is necessary to have a two-level recovery mechanism. A **global recovery manager**, or **coordinator**, is needed in addition to the local recovery managers. The coordinator usually follows a protocol called the **two-phase commit protocol**, whose two phases can be stated as follows:

PHASE 1: When all participating databases signal the coordinator that the part of the multidatabase transaction involving each has concluded, the coordinator sends a message “prepare for commit” to each participant to get ready for committing the transaction. Each participating database receiving that message will force-write all log records to disk and then send a “ready to commit” or “OK” signal to the coordinator. If the force-writing to disk fails or the local transaction cannot commit for some reason, the participating database sends a “cannot commit” or “not OK” signal to the coordinator. If the coordinator does not receive a reply from a database within a certain time interval, it assumes a “not OK” response.

PHASE 2: If *all* participating databases reply “OK,” the transaction is successful, and the coordinator sends a “commit” signal for the transaction to the participating databases. Because all the local effects of the transaction have been recorded in the logs of the participating databases, recovery from failure is now possible. Each participating database completes transaction commit by writing a [commit] entry for the transaction in the log and permanently updating the database if needed. On the other hand, if one or more of the participating databases have a “not OK” response to the coordinator, the transaction has failed, and the coordinator sends a message to “roll back” or UNDO the local effect of the transaction to each participating database. This is done by undoing the transaction operations, using the log. ■

The net effect of the two-phase commit protocol is that either all participating databases commit the effect of the transaction or none of them do. In case any of the participants—or the coordinator—fails, it is always possible to recover to a state where either the transaction is committed or it is rolled back. A failure during or before phase 1 usually requires the transaction to be rolled back, whereas a failure during phase 2 means that a successful transaction can recover and commit.

19.6 Database Backup and Recovery from Catastrophic Failures

So far, all the techniques we have discussed apply to noncatastrophic failures. A key assumption has been that the system log is maintained on the disk and is not lost as a result of the failure. Similarly, the shadow page table must be stored on disk to allow recovery when shadow paging is used. The recovery techniques we have discussed use the entries in the system log or the shadow page table to recover from the failure by bringing the database back to a consistent state.

The recovery manager of a DBMS must also be equipped to handle more catastrophic failures such as disk crashes. The main technique used to handle such crashes is that of **database backup**. The whole database and the log are periodically copied onto a cheap storage medium such as magnetic tapes. In case of a catastrophic system failure, the latest backup copy can be reloaded from the tape to the disk, and the system can be restarted.

To avoid losing all the effects of transactions that have been executed since the last backup, it is customary to back up the system log by periodically copying it to magnetic tape. The system log is usually substantially smaller than the database itself and hence can be backed up more frequently. When the system log is backed up, users do not lose all transactions they have performed since the last database backup. All committed transactions recorded in the portion of the system log that has been backed up can have their effect on the database reconstructed. A new system log is started after each database backup operation. Hence, to recover from disk failure, the database is first re-created on disk from its latest backup copy on tape. Following that, the effects of all the committed transactions whose operations have been entered in the backed-up copy of the system log are reconstructed.

19.7 Summary

In this chapter we discussed the techniques for recovery from transaction failures. The main goal of recovery is to ensure the atomicity property of a transaction. If a transaction fails before completing its execution, the recovery mechanism has to make sure that the transaction has no lasting effects on the database. We first gave an informal outline for a recovery process and then discussed system concepts for recovery. These included a discussion of caching, in-place updating versus shadowing, before and after images of a data item, UNDO versus REDO recovery operations, and the write-ahead logging protocol.

Next we discussed two different approaches to recovery: deferred update and immediate update. Deferred update techniques postpone any actual updating of the database until a transaction reaches its commit point. The transaction force-writes the log to disk

before recording the updates in the database. This approach, when used with certain concurrency control methods, is designed never to require transaction rollback, and recovery simply consists of redoing the operations of transactions committed after the last checkpoint from the log. Deferred update can lead to a recovery algorithm known as NO-UNDO/REDO. Immediate update techniques may apply changes to the database before the transaction reaches a successful conclusion. In some protocols, any changes applied to the database must first be recorded in the log and force-written to disk so that these operations can be undone if necessary. We also gave an overview of a recovery algorithm for immediate update known as UNDO/REDO. Another algorithm, known as UNDO/NO-REDO, can also be developed for immediate update if all transaction actions are recorded in the database before commit.

We discussed the shadow paging technique for recovery, which keeps track of old database pages by using a shadow page table. This technique, which is classified as NO-UNDO/NO-REDO, does not require a log in single-user systems but still needs the log for multiuser systems. We then discussed the two-phase commit protocol, which is used for recovery from failures involving multidatabase transactions. Finally, we discussed recovery from catastrophic failures, which is typically done by backing up the database and the log to tape. The log can be backed up more frequently than the database, and the backup log can be used to redo operations starting from the last database backup.

Review Questions

- 19.1. Discuss the different types of transaction failures. What is meant by catastrophic failure?
- 19.2. Discuss the actions taken by the read_item and write_item operations on a database.
- 19.3. (Review from Chapter 17) What is the system log used for? What are the typical kinds of entries in a system log? What are checkpoints, and why are they important? What are transaction commit points, and why are they important?
- 19.4. How are buffering and caching techniques used by the recovery subsystem?
- 19.5. What are the before image (BFIM) and the after image (AFIM) of a data item? What is the difference between in-place updating and shadowing, with respect to their handling of BFIM and AFIM?
- 19.6. What are UNDO-type log entries and REDO-type log entries?
- 19.7. Describe the write-ahead logging protocol.
- 19.8. Identify three typical lists of transactions that are maintained by the recovery subsystem.
- 19.9. What is meant by transaction rollback? Why is it necessary to check for cascading rollback? Which recovery techniques do not require rollback?
- 19.10. Discuss the UNDO and REDO operations and the recovery techniques that use each.
- 19.11. Discuss the deferred update technique of recovery. What is the main advantage of this technique? Why is it called the NO-UNDO/REDO method?

- 19.12. How can recovery handle transaction operations that do not affect the database, such as the printing of reports by the transaction?
- 19.13. Discuss the immediate update recovery technique in both single-user and multiuser environments. What are the advantages and disadvantages of immediate update?
- 19.14. What is the difference between the UNDO/REDO and the UNDO/NO-REDO algorithms for recovery with immediate update? Develop the outline for an UNDO/NO-REDO algorithm.
- 19.15. Describe the shadow paging recovery technique. Under what circumstances does it not require a log?
- 19.16. Describe the two-phase commit protocol for multidatabase transactions.
- 19.17. Discuss how recovery from catastrophic failures is handled.

Exercises

- 19.18. Suppose that the system crashes before the [read_item, T₃, A] entry is written to the log in Figure 19.1(b); will that make any difference in the recovery process?
- 19.19. Suppose that the system crashes before the [write_item, T₂, D, 25, 26] entry is written to the log in Figure 19.1(b); will that make any difference in the recovery process?
- 19.20. Figure 19.6 shows the log corresponding to a particular schedule at the point of a system crash for the four transactions T₁, T₂, T₃, and T₄ of Figure 19.4. Suppose

```

[start_transaction, T1]
[read_item, T1, A]
[read_item, T1, D]
[write_item, T1, D, 20]
[commit, T1]
[checkpoint]
[start_transaction, T2]
[read_item, T2, B]
[write_item, T2, B, 12]
[start_transaction, T4]
[read_item, T4, B]
[write_item, T4, B, 15]
[start_transaction, T3]
[write_item, T3, A, 30]
[read_item, T4, A]
[write_item, T4, A, 20]
[commit, T4]
[read_item, T2, D]
[write_item, T2, D, 25] ← System crash

```

Figure 19.6 Example schedule and corresponding log.

- that we use the *immediate update protocol* with checkpointing. Describe the recovery process from the system crash. Specify which transactions are rolled back, which operations in the log are redone and which (if any) are undone, and whether any cascading rollback takes place.
- 19.21. Suppose that we use the deferred update protocol for the example in Figure 19.6. Show how the log would be different in the case of deferred update by removing the unnecessary log entries; then describe the recovery process, using your modified log. Assume that only redo operations are applied, and specify which operations in the log are redone and which are ignored.

Selected Bibliography

The books by Bernstein et al. (1987) and Papadimitriou (1986) are devoted to the theory and principles of concurrency control and recovery. The book by Gray and Reuter (1993) is an encyclopedic work on concurrency control, recovery, and other transaction-processing issues.

Verhofstad (1978) presents a tutorial and survey of recovery techniques in database systems. Categorizing algorithms based on their UNDO/REDO characteristics is discussed in Haerder and Reuter (1983) and in Bernstein et al. (1983). Gray (1978) discusses recovery, along with other system aspects of implementing operating systems for databases. The shadow paging technique is discussed in Lorie (1977), Verhofstad (1978), and Reuter (1980). Gray et al. (1981) discusses the recovery mechanism in SYSTEM R. Lockeman and Knutsen (1968), Davies (1972), and Bjork (1973) are early papers that discuss recovery. Chandy et al. (1975) discusses transaction rollback. Lilien and Bhargava (1985) discusses the concept of integrity block and its use to improve the efficiency of recovery.

Recovery using write-ahead logging is analyzed in Jhingran and Khedkar (1992) and is used in the ARIES system (Mohan et al. 1992a). More recent work on recovery includes compensating transactions (Korth et al. 1990) and main memory database recovery (Kumar 1991). The ARIES recovery algorithms (Mohan et al. 1992) have been quite successful in practice. Franklin et al. (1992) discusses recovery in the EXODUS system.

CHAPTER 23

Distributed Databases and Client–Server Architecture

In a **centralized database system**, all system components reside at a single computer or site. The components include the data, the DBMS software, and the associated secondary storage devices such as disks for on-line database storage and tapes for backup. A centralized database can be accessed remotely via terminals connected to the site; however, the data and DBMS software principally reside at a single site. In recent years there has been a rapid trend toward the **distribution** of computer systems over multiple sites that are interconnected via a **communication network**. In this chapter we discuss the development of **distributed database systems (DDBSs)** and the techniques used in their implementation. The software used to implement such a system is called a **distributed database management system (DDBMS)**.

In Section 23.1 we discuss the reasons for moving toward distribution, and we introduce DDBS concepts. Section 23.2 gives an overview of the client–server architecture, which is becoming popular for distributed systems in general, and for distributed databases in particular. Section 23.3 presents techniques for fragmenting a database and distributing its portions over multiple sites; this is also known as distributed database design. Section 23.4 categorizes the various types of distributed database systems. Sections 23.5 and 23.6 introduce distributed query processing and concurrency control/recovery techniques, respectively.

For a shorter introduction to the topic of distributed databases, some or all of Sections 23.3.3, 23.5, and 23.6 may be skipped.

23.1 Introduction to Distributed DBMS Concepts

A **distributed database** is a collection of data that belongs logically to the same system but is physically spread over the sites of a computer network. Several factors have led to the development of DDBSs. Potential advantages of DDBSs include the following:

- *Distributed nature of some database applications*: Many database applications are *naturally distributed* over different locations. For example, a company may have locations at different cities, or a bank may have multiple branches. It is natural for databases used in such applications to be distributed over these locations. Many **local users** access only the data at this location, but other **global users**—such as company headquarters—may require occasional access to data stored at several of these locations. Notice that the data at each local site typically describes a “mini-world” at that site. The sources of data and the majority of users and applications for the local database physically reside at that site.
- *Increased reliability and availability*: These are two of the most common potential advantages cited for distributed databases. **Reliability** is broadly defined as the probability that a system is up at a particular moment, whereas **availability** is the probability that the system is continuously available during a time interval. When the data and DBMS software are distributed over several sites, one site may fail while other sites continue to operate. Only the data and software that exist at the failed site cannot be accessed. This improves both reliability and availability. Further improvement is achieved by judiciously *replicating* data and software at more than one site. In a centralized system, failure at a single site makes the whole system unavailable to all users.
- *Allowing data sharing while maintaining some measure of local control*: In some types of DDBSs (see Section 23.4), it is possible to control the data and software locally at each site. However, certain data can be accessed by users at other remote sites through the DDBMS software. This allows *controlled* sharing of data throughout the distributed system.
- *Improved performance*: When a large database is distributed over multiple sites, smaller databases exist at each site. As a result, local queries and transactions accessing data at a single site have better performance because of the smaller local databases. In addition, each site has a smaller number of transactions executing than if all transactions are submitted to a single centralized database. For transactions that involve access to more than one site, processing at the different sites may proceed in parallel, reducing response time.

Distribution leads to increased complexity in the system design and implementation. To achieve the potential advantages listed previously, the DDBMS software must be able to provide the following *additional functions* to those of a centralized DBMS:

- The ability to access remote sites and transmit queries and data among the various sites via a communication network.
- The ability to keep track of the data distribution and replication in the DDBMS catalog.

- The ability to devise execution strategies for queries and transactions that access data from more than one site.
- The ability to decide on which copy of a replicated data item to access.
- The ability to maintain the consistency of copies of a replicated data item.
- The ability to recover from individual site crashes and from new types of failures such as the failure of a communication link.

These functions themselves increase the complexity of a DDBMS over a centralized DBMS. Before we can realize the full potential advantages of distribution, we must find satisfactory solutions to these issues. Including all this additional functionality is hard to accomplish, and finding optimal solutions is a step beyond that. Additional complexities appear when we consider the design of a distributed database. Specifically, we must decide on how to distribute the data over sites and what data, if any, to replicate.

At the physical **hardware** level, the following main factors distinguish a DDBS from a centralized system:

- There are multiple computers, called **sites** or **nodes**.
- These sites must be connected by some type of **communication network** to transmit data and commands among sites, as shown in Figure 23.1.

The sites may all be located in physical proximity—say, within the same building or group of adjacent buildings—and connected via a **local area network**, or they may be geographically distributed over large distances and connected via a **long-haul net-**

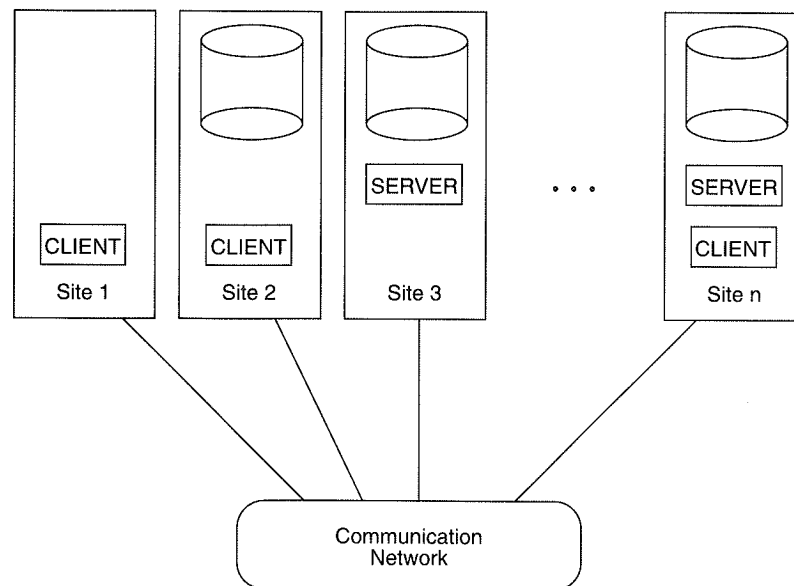


Figure 23.1 Simplified physical client-server architecture for a DDBS.

work. Local area networks typically use cables, whereas long-haul networks use telephone lines or satellites. It is also possible to use a combination of the two types of networks.

Networks may have different **topologies** that define the direct communication paths among sites. For example, direct links may exist between sites 1 and 2 and between sites 2 and 3 but not between sites 1 and 3; in such a case, communication between sites 1 and 3 must pass through site 2. The type and topology of the network used may have a significant effect on performance and hence on the strategies for distributed query processing and distributed database design. For high-level architectural issues, however, it does not matter which type of network is used; it only matters that each site is able to communicate, directly or indirectly, with every other site. For the remainder of this chapter, we assume that some type of communication network exists among sites, regardless of the particular topology.

23.2 Overview of Client-Server Architecture

The **client-server architecture** has been developed to deal with new computing environments in which a large number of personal computers, workstations, file servers, printers, and other equipment are connected together via a network. The idea is to define **specialized servers** with specific functionalities. For example, it is possible to connect a number of diskless workstations or personal computers as clients to a *file server* machine that maintains the files of the client's users. Another machine could be designated as a *printer server* by being connected to various printers; thereafter, all print requests would be forwarded to this machine. In this way, the resources provided by specialized servers can be accessed by many clients. This idea can be carried over to software, with specialized software—such as a DBMS or a CAD (computer-aided design) package—being stored on specific server machines, and this software being made accessible to multiple clients.

The client-server architecture is increasingly being incorporated into commercial DBMS packages as they move toward the support of distribution. The idea is to divide the DBMS software into two levels—client and server—to reduce its complexity. This is illustrated in Figure 23.1; some sites may run the client software only (these could be diskless machines, such as site 1, or machines with disks, such as site 2), while other sites may be dedicated server machines that run the server software only, such as site 3. Still other sites may support both client and server modules, such as site *n* in Figure 23.1.

Exactly how to divide the DBMS functionality between client and server has not yet been established. Different approaches have been proposed. One possibility is to include the functionality of a centralized DBMS at the server level. A number of relational DBMS products have taken this approach, where an **SQL server** is provided to the clients. Each client must then formulate the appropriate SQL queries and provide the user interface and programming language interface functions. Since SQL is a relational standard, various different SQL servers, possibly provided by different vendors, can accept SQL commands. The client may also refer to a data dictionary that includes information on the distribution of data among the various SQL servers, as well as modules for decomposing a global

query into a number of local queries that can be executed at the various sites. We discuss such a module further in Section 23.5. Interaction between client and server might proceed as follows during processing of an SQL query:

1. The client parses a user query and decomposes it into a number of independent site queries. Each site query is sent to the appropriate server site.
2. Each server processes the local query and sends the resulting relation to the client site.
3. The client site combines the results of the subqueries to produce the result of the originally submitted query.

In this approach, the SQL server has also been called a **database processor (DP)** or a **back-end machine**, whereas the client has been called an **application processor (AP)** or a **front-end machine**. The interaction between client and server can be specified by the user at the client level or via a specialized client module. For example, the user may know what data is stored in each server, break down a query request into site subqueries manually, and submit individual subqueries to the various sites. The resulting tables may be combined explicitly by a further user query at the client level. The alternative is to have the client module undertake these actions automatically.

Another approach, taken by some object-oriented DBMSs (see Chapter 22), divides the software modules of the DBMS between client and server in a more integrated way. For example, the *server level* may include the part of the DBMS software responsible for handling data storage on disk pages, local concurrency control and recovery, buffering and caching of disk pages, and other such functions. Meanwhile, the *client level* may handle the user interface, data dictionary functions, DBMS interaction with programming language compilers, global query optimization/concurrency control/recovery, structuring of complex objects from the data in the buffers, and other such functions. In this approach, the client-server interaction is more tightly coupled and is done internally by the DBMS modules, rather than by the users.

In a typical DDBMS, it is customary to divide the software modules into three levels:

- The **server** software is responsible for local data management at a site, much like centralized DBMS software.
- The **client** software is responsible for most of the distribution functions; it accesses data distribution information from the DDBMS catalog and processes all requests that require access to more than one site.
- The **communications software** (sometimes in conjunction with a **distributed operating system**) provides the communication primitives that are used by the client to transmit commands and data among the various sites as needed. This is not strictly part of the DDBMS, but it provides essential communication primitives and services.

The **client** is responsible for generating a distributed execution plan for a multisite query or transaction and for supervising distributed execution by sending commands to servers. These commands include local queries and transactions to be executed, as well as commands to transmit data to other clients or servers. Hence, client software should be included at any site where multisite queries are submitted. We discuss distributed

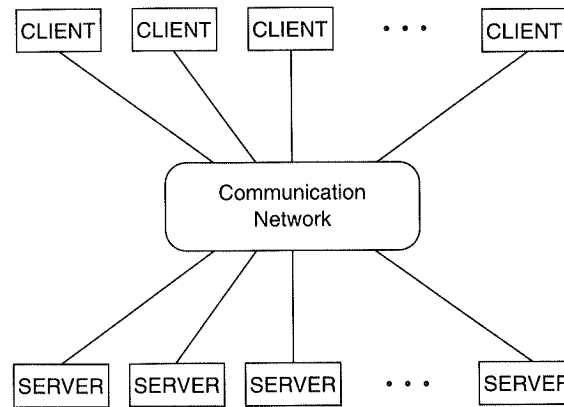


Figure 23.2 Simplified logical client-server architecture for a DDBS.

query processing further in Section 23.5. Another function controlled by the client is that of ensuring consistency of replicated copies of a data item by employing distributed (or global) concurrency control techniques. The client must also ensure the atomicity of global transactions by performing global recovery when certain sites fail. We discuss distributed recovery and concurrency control in Section 23.6. Figure 23.2 shows a logical view of a DDBS; here, clients and servers are shown without specifying the site on which each resides.

One possible function of the client is to *hide* the details of data distribution from the user; that is, it enables the user to write global queries and transactions as though the database were centralized, without having to specify the sites at which the data referenced in the query or transaction resides. This property is called **distribution transparency**. Some DDBMSs do not provide distribution transparency, instead requiring that users be aware of the details of data distribution. In this case, users must specify the sites at which the data referenced in global queries and transactions reside. In the latter situation, a user typically *appends the site name* to any references to a relation, file, or record type. In the former situation, the user is presented with a schema that does not include any distribution information, and the DDBMS itself keeps track of the sites at which data is located in the **DDBMS catalog**. Using this information, the client software can break down a query into a number of subqueries that can be executed at the various sites, and it can plan how to transmit results of subqueries to other sites for further processing and for producing the result. DDBMSs that provide distribution transparency make it simpler for a user to specify queries and transactions, but they require more complex software. DDBMSs that do not provide distribution transparency make it the responsibility of the user to specify the site of each relation or file, leading to simpler software.

23.3 Data Fragmentation, Replication, and Allocation

Techniques for Distributed Database Design

In this section we discuss techniques that are used to break up the database into logical units, called **fragments**, that may be assigned for storage at the various sites. We also discuss the use of **data replication**, which permits certain data to be stored in more than one site, and the process of **allocating** fragments—or replicas of fragments—for storage at the various sites. These techniques are used during the process of **distributed database design**. The information concerning data fragmentation, allocation, and replication is stored in a **global system catalog** that is accessed by the client software as needed.

23.3.1 Data Fragmentation

In a DDBS, decisions must be made regarding which site should be used to store which portions of the database. For now, we will assume that there is *no replication*; that is, each file—or portion of a file—is to be stored at only one site. We discuss replication and its effects later in this section. We also use the terminology of *relational databases*; similar concepts apply to other data models. We assume that we are starting with a relational database schema and must decide on how to distribute the relations over the various sites. To illustrate our discussion, we use the relational database schema in Figure 6.5.

Before we decide on how to distribute the data, we must determine the *logical units* of the database that are to be distributed. The simplest logical units are the relations themselves; that is, each *whole* relation is to be stored at a particular site. In our example, we must decide on a site to store each of the relations EMPLOYEE, DEPARTMENT, PROJECT, WORKS_ON, and DEPENDENT of Figure 6.5. In many cases, however, a relation can be divided into smaller logical units for distribution. For example, consider the company database shown in Figure 6.6, and assume there are three computer sites—one for each department in the company. Of course, in an actual situation, there are many more tuples in the relations. We may want to store the database information relating to each department at the computer site for that department. To do so, we need to partition each relation by using a technique called horizontal fragmentation.

Horizontal Fragmentation. A **horizontal fragment** of a relation is a subset of the tuples in that relation. The tuples that belong to the horizontal fragment are specified by a condition on one or more attributes of the relation. Often, only a single attribute is involved. For example, we may define three horizontal fragments on the EMPLOYEE relation of Figure 6.6 with the following conditions: (DNO = 5), (DNO = 4), and (DNO = 1); each fragment contains the EMPLOYEE tuples working for a particular department. Similarly, we may define three horizontal fragments for the PROJECT relation of Figure 6.6 with the conditions (DNUM = 5), (DNUM = 4), and (DNUM = 1); each fragment contains the PROJECT tuples controlled by a particular department. Horizontal fragmentation divides a relation “horizontally” by grouping rows to create subsets of tuples, where each subset has a certain logical meaning. These fragments can then be assigned to different sites in the distributed system.

Vertical Fragmentation. A **vertical fragment** of a relation keeps only certain attributes of the relation. For example, we may want to fragment the EMPLOYEE relation into two vertical fragments. The first fragment includes personal information—NAME, BDATE, ADDRESS, and SEX—and the second includes work-related information—SSN, SALARY, SUPERSSN, DNO. This vertical fragmentation is not quite proper because, if the two fragments are stored separately, we cannot put the original employee tuples back together, since there is *no common attribute* between the two fragments. It is necessary to include the *primary key attribute* in *every* vertical fragment so that the full relation can be reconstructed from the fragments. Hence, we must add the SSN attribute to the personal information fragment. Vertical fragmentation divides a relation “vertically” by columns.

Notice that each horizontal fragment on a relation R can be specified by a $\sigma_{C_i}(R)$ operation in the relational algebra. A set of horizontal fragments whose conditions $C_1, C_2, \dots, C_n$ include all the tuples in R —that is, every tuple in R satisfies $(C_1 \text{ OR } C_2 \text{ OR } \dots \text{ OR } C_n)$ —is called a **complete horizontal fragmentation** of R . In many cases a complete horizontal fragmentation is also **disjoint**; that is, no tuple in R satisfies $(C_i \text{ AND } C_j)$ for any $i \neq j$. Our two earlier examples of horizontal fragmentation for the EMPLOYEE and PROJECT relations were both complete and disjoint. To reconstruct the relation R from a *complete* horizontal fragmentation, we need to apply the UNION operation to the fragments.

A vertical fragment on a relation R can be specified by a $\pi_{L_i}(R)$ operation in the relational algebra. A set of vertical fragments whose projection lists $L_1, L_2, \dots, L_n$ include all the attributes in R but share only the primary key attribute of R is called a **complete vertical fragmentation** of R . In this case the projection lists satisfy the following two conditions:

- $L_1 \cup L_2 \cup \dots \cup L_n = \text{ATTRS}(R)$.
- $L_i \cap L_j = \text{PK}(R)$ for any $i \neq j$, where $\text{ATTRS}(R)$ is the set of attributes of R and $\text{PK}(R)$ is the primary key of R .

To reconstruct the relation R from a *complete* vertical fragmentation, we apply the OUTER UNION operation to the fragments. Notice that we could also apply the FULL OUTER JOIN operation and get the same result for a complete vertical fragmentation. The two vertical fragments of the EMPLOYEE relation with projection lists $L_1 = \{\text{SSN, NAME, BDATE, ADDRESS, SEX}\}$ and $L_2 = \{\text{SSN, SALARY, SUPERSSN, DNO}\}$ constitute a complete vertical fragmentation of EMPLOYEE.

Two horizontal fragments that are neither complete nor disjoint are those defined on the EMPLOYEE relation of Figure 6.5 by the conditions $(\text{SALARY} > 50000)$ and $(\text{DNO} = 4)$; they may not include all EMPLOYEE tuples, and they may include common tuples. Two vertical fragments that are not complete are those defined by the attribute lists $L_1 = \{\text{NAME, ADDRESS}\}$ and $L_2 = \{\text{SSN, NAME, SALARY}\}$; these lists violate both conditions of a complete vertical fragmentation.

Mixed Fragmentation. We can intermix the two types of fragmentation, yielding a **mixed fragmentation**. For example, we may combine the horizontal and vertical fragmentations of the EMPLOYEE relation given earlier into a mixed fragmentation that includes six fragments. In this case the original relation can be reconstructed by applying

UNION and OUTER UNION (or OUTER JOIN) operations in the appropriate order. In general, a **fragment** of a relation R can be specified by a SELECT-PROJECT combination of operations $\pi_L(\sigma_C(R))$. If $C = \text{TRUE}$ and $L \neq \text{ATTRS}(R)$, we get a vertical fragment, and if $C \neq \text{TRUE}$ and $L = \text{ATTRS}(R)$, we get a horizontal fragment. Finally, if $C \neq \text{TRUE}$ and $L \neq \text{ATTRS}(R)$, we get a mixed fragment. Notice that a relation can itself be considered a fragment with $C = \text{TRUE}$ and $L = \text{ATTRS}(R)$. In the following discussion, the term *fragment* is used to refer to a relation or to any of the preceding types of fragments.

A **fragmentation schema** of a database is a definition of a set of fragments that includes *all* attributes and tuples in the database and satisfies the condition that the whole database can be reconstructed from the fragments by applying some sequence of OUTER UNION (or OUTER JOIN) and UNION operations. It is also sometimes useful—although not necessary—to have all the fragments be disjoint except for the repetition of primary keys among vertical (or mixed) fragments. In the latter case, all replication and distribution of fragments is clearly specified at a subsequent stage, separately from fragmentation.

An **allocation schema** describes the allocation of fragments to sites of the DDBS; hence, it is a mapping that specifies for each fragment the site(s) at which it is stored. If a fragment is stored at more than one site, it is said to be **replicated**. We discuss data replication and allocation next.

23.3.2 Data Replication and Allocation

Replication is useful in improving the availability of data. The most extreme case is replication of the *whole database* at every site in the distributed system, thus creating a **fully replicated** distributed database. This can improve availability remarkably because the system can continue to operate as long as at least one site is up. It also improves performance of retrieval for global queries, because the result of such a query can be obtained locally from any one site; hence, a retrieval query can be processed at the local site where it is submitted, if that site includes a server module. The disadvantage of full replication is that it can slow down update operations drastically, since a single logical update must be performed on every copy of the database to keep the copies consistent. This is especially true if many copies of the database exist. Full replication makes the concurrency control and recovery techniques more expensive than they would be if there were no replication, as we shall see in Section 23.6.

The other extreme from full replication involves having **no replication**; that is, each fragment is stored at exactly one site. In this case all fragments *must* be disjoint, except for the repetition of primary keys among vertical (or mixed) fragments. This is also called **nonredundant allocation**.

Between these two extremes, we have a wide spectrum of **partial replication** of the data; that is, some fragments of the database may be replicated whereas others are not. The number of copies of each fragment can range from one up to the total number of sites in the distributed system. A description of the replication of fragments is sometimes called a **replication schema**.

Each fragment—or each copy of a fragment—must be assigned to a particular site in the distributed system. This process is called **data distribution** (or **data allocation**).

The choice of sites and the degree of replication depend on the performance and availability goals of the system and on the types and frequencies of transactions submitted at each site. For example, if high availability is required and if transactions can be submitted at any site and if most transactions are retrieval only, a fully replicated database is a good choice. However, if certain transactions that access particular parts of the database are mostly submitted at a particular site, the corresponding set of fragments can be allocated at that site only. Data that is accessed at multiple sites can be replicated at those sites. If many updates are performed, it may be useful to limit replication. Finding an optimal or even a good solution to distributed data allocation is a complex optimization problem.

23.3.3 Example of Fragmentation, Allocation, and Replication★

We now consider an example of fragmenting and distributing the company database of Figures 6.5 and 6.6. Suppose that the company has three computer sites—one for each current department. Sites 2 and 3 are for departments 5 and 4, respectively. At each of these sites, we expect frequent access to the `EMPLOYEE` and `PROJECT` information for the employees *who work in that department* and the projects *controlled by that department*. Further, we assume that these sites mainly access the `NAME`, `SSN`, `SALARY`, and `SUPERSSN` attributes of `EMPLOYEE`. Site 1 is used by company headquarters and accesses all employee and project information regularly, in addition to keeping track of `DEPENDENT` information for insurance purposes.

According to these requirements, the whole database of Figure 6.6 can be stored at site 1. To determine the fragments to be replicated at sites 2 and 3, we can first horizontally fragment the `EMPLOYEE`, `PROJECT`, `DEPARTMENT`, and `DEPT_LOCATIONS` relations by department number—called `DNO`, `DNUM`, and `DNUMBER`, respectively, in Figure 6.5. We can then vertically fragment the resulting `EMPLOYEE` fragments to include only the attributes {`NAME`, `SSN`, `SALARY`, `SUPERSSN`, `DNO`}. Figure 23.3 shows the mixed fragments `EMPD5` and `EMPD4`, which include the `EMPLOYEE` tuples satisfying the conditions `DNO = 5` and `DNO = 4`, respectively. The horizontal fragments of `PROJECT`, `DEPARTMENT`, and `DEPT_LOCATIONS` are similarly fragmented by department number. All these fragments—stored at sites 2 and 3—are replicated because they are also stored at the headquarters site 1.

We must now fragment the `WORKS_ON` relation and decide which fragments of `WORKS_ON` to store at sites 2 and 3. We are confronted with the problem that no attribute of `WORKS_ON` directly indicates the department to which each tuple belongs. In fact, each tuple in `WORKS_ON` relates an employee *e* to a project *p*. We could fragment `WORKS_ON` based on the department *d* in which *e* works or based on the department *d'* that controls *p*. Fragmentation becomes easy if we have a constraint stating that $d = d'$ for all `WORKS_ON` tuples—that is, if employees can work only on projects controlled by the department they work for. However, there is no such constraint in our database of Figure 6.6. For example, the `WORKS_ON` tuple `<333445555, 10, 10.0>` relates an employee who works for department 5 with a project controlled by department 4. In this case we could fragment `WORKS_ON` based on *both* the department in which the employee works *and* the department that controls the project, as shown in Figure 23.4.

In Figure 23.4, the union of fragments `G1`, `G2`, and `G3` gives all `WORKS_ON` tuples for employees who work *for department 5*. Similarly, the union of fragments `G4`, `G5`, and

(a)

EMPID5	FNAME	MINIT	LNAME	SSN	SALARY	SUPERSSN	DNO
	John	B	Smith	123456789	30000	333445555	5
	Franklin	T	Wong	333445555	40000	888665555	5
	Ramesh	K	Narayan	666884444	38000	333445555	5
	Joyce	A	English	453453453	25000	333445555	5

DEP5	DNAME	DNUMBER	MGRSSN	MGRSTARTDATE
	Research	5	333445555	22- MAY-78

DEP5_LOCS	DNUMBER	LOCATION
	5	Bellaire
	5	Sugarland
	5	Houston

WORKS_ON5	ESSN	PNO	HOURS
	123456789	1	32.5
	123456789	2	7.5
	666884444	3	40.0
	453453453	1	20.0
	453453453	2	20.0
	333445555	2	10.0
	333445555	3	10.0
	333445555	10	10.0
	333445555	20	10.0

PROJS5	PNAME	PNUMBER	PLOCATION	DNUM
	Product X	1	Bellaire	5
	Product Y	2	Sugarland	5
	Product Z	3	Houston	5

(b)

EMPID4	FNAME	MINIT	LNAME	SSN	SALARY	SUPERSSN	DNO
	Alicia	J	Zelaya	999887777	25000	987654321	4
	Jennifer	S	Wallace	987654321	43000	888665555	4
	Ahmad	V	Jabbar	987987987	25000	987654321	4

DEP4	DNAME	DNUMBER	MGRSSN	MGRSTARTDATE
	Administration	4	987654321	01-JAN-85

DEP4_LOCS	DNUMBER	LOCATION
	4	Stafford

WORKS_ON4	ESSN	PNO	HOURS
	333445555	10	10.0
	999887777	30	30.0
	999887777	10	10.0
	987987987	10	35.0
	987987987	30	5.0
	987654321	30	20.0
	987654321	20	15.0

PROJS4	PNAME	PNUMBER	PLOCATION	DNUM
	Computerization	10	Stafford	4
	Newbenefits	30	Stafford	4

Figure 23.3 Allocation of fragments to sites. (a) Relation fragments at site 2.
(b) Relation fragments at site 3.

(a)

G1	ESSN	PNO	HOURS
	123456789	1	32.5
	123456789	2	7.5
	666884444	3	40.0
	453453453	1	20.0
	453453453	2	20.0
	333445555	2	10.0
	333445555	3	10.0

C1=C AND (PNO IN (SELECT PNUMBER
FROM PROJECT
WHERE DNUM=5))

G2	ESSN	PNO	HOURS
	333445555	10	10.0

C2=C AND (PNO IN (SELECT PNUMBER
FROM PROJECT
WHERE DNUM=4))

G3	ESSN	PNO	HOURS
	333445555	20	10.0

C3=C AND (PNO IN (SELECT PNUMBER
FROM PROJECT
WHERE DNUM=1))

(b)

G4	ESSN	PNO	HOURS
----	------	-----	-------

C4=C AND (PNO IN (SELECT PNUMBER
FROM PROJECT
WHERE DNUM=5))

G5	ESSN	PNO	HOURS
	999887777	30	30.0
	999887777	10	10.0
	987987987	10	35.0
	987987987	30	5.0
	987654321	30	20.0

C5=C AND (PNO IN (SELECT PNUMBER
FROM PROJECT
WHERE DNUM=4))

G6	ESSN	PNO	HOURS
	987654321	20	15.0

C6=C AND (PNO IN (SELECT PNUMBER
FROM PROJECT
WHERE DNUM=1))

(c)

G7	ESSN	PNO	HOURS
----	------	-----	-------

C7=C AND (PNO IN (SELECT PNUMBER
FROM PROJECT
WHERE DNUM=5))

G8	ESSN	PNO	HOURS
----	------	-----	-------

C8=C AND (PNO IN (SELECT PNUMBER
FROM PROJECT
WHERE DNUM=4))

G9	ESSN	PNO	HOURS
	888665555	20	null

C9=C AND (PNO IN (SELECT PNUMBER
FROM PROJECT
WHERE DNUM=1))

Figure 23.4 Complete and disjoint fragments of the WORKS_ON relation.
 (a) Fragments of WORKS_ON for employees working in department 5 ($C=ESSN \text{ IN } (SELECT SSN \text{ FROM } EMPLOYEE \text{ WHERE } DNO=5)$). (b) Fragments of WORKS_ON for employees working in department 4 ($C=ESSN \text{ IN } (SELECT SSN \text{ FROM } EMPLOYEE \text{ WHERE } DNO=4)$). (c) Fragments of WORKS_ON for employees working in department 1 ($C=ESSN \text{ IN } (SELECT SSN \text{ FROM } EMPLOYEE \text{ WHERE } DNO=1)$).

G6 gives all WORKS_ON tuples for employees who *work for department 4*. On the other hand, the union of fragments G1, G4, and G7 gives all WORKS_ON tuples for projects *controlled by department 5*. The condition for each of the fragments G1 through G9 is shown in Figure 23.4. The relations that represent M:N relationships, such as WORKS_ON, often have several possible logical fragmentations. In our distribution of Figure 23.3, we choose to include all fragments that can be joined to either an EMPLOYEE tuple or a PROJECT tuple at sites 2 and 3. Hence, we place the union of fragments G1, G2, G3, G4, and G7 at site 2 and the union of fragments G4, G5, G6, G2, and G8 at site 3. Notice that fragments G2 and G4 are replicated at both sites. This allocation

strategy permits the join between the local EMPLOYEE or PROJECT fragments at site 2 or site 3 and the local WORKS_ON fragment to be performed completely locally. This clearly demonstrates how complex the problem of database fragmentation and allocation is for large databases. The bibliographic notes give references to some of the work done in this area.

23.4 Types of Distributed Database Systems

The term *distributed database management system* can describe various systems that differ from one another in many respects. The main thing that all such systems have in common is the fact that data and software are distributed over multiple sites connected by some form of communication network. In this section we discuss a number of types of DDBMSs and the criteria and factors that make some of these systems different.

The first factor we consider is the **degree of homogeneity** of the DDBMS software. If all servers (or individual local DBMSs) use identical software and all clients use identical software, the DDBMS is called **homogeneous**; otherwise, it is called **heterogeneous**. Another factor related to the degree of homogeneity is the **degree of local autonomy**. If all access to the DDBMS must be gained through a client, then the system has **no local autonomy**. On the other hand, if *direct access* by local transactions to a server is permitted, the system has some degree of local autonomy.

At one extreme of the autonomy spectrum, we have a DDBMS that “looks like” a centralized DBMS to the user. A single conceptual schema exists, and all access to the system is obtained through a client, so no local autonomy exists. At the other extreme we encounter a type of DDBMS called a **federated DDBMS** (or a **multidatabase system**). In such a system, each server is an independent and autonomous centralized DBMS that has its own local users, local transactions, and DBA and hence has a very high degree of *local autonomy*. Each server can authorize access to particular portions of its database by specifying an **export schema**, which specifies the part of the database that may be accessed by a certain class of nonlocal users. A client in such a system is essentially an additional interface to several servers (local DBMSs) that allows a multidatabase (or global) user to access data stored in several of these *autonomous databases*. Notice that a federated system is a hybrid between distributed and centralized systems; it is a centralized system for local autonomous users and a distributed system for global users.

In a heterogeneous multidatabase system, one server may be a relational DBMS, another a network DBMS, and a third a hierarchical DBMS; in such a case it is necessary to have a canonical system language and to include language translators in the client to translate subqueries from the canonical language to the language of each server.

A third aspect that may be used to categorize distributed databases is the **degree of distribution transparency** or, alternatively, the **degree of schema integration**. If the user sees a single integrated schema without any information concerning fragmentation, replication, or distribution, the DDBMS is said to have a *high degree of distribution transparency* (or schema integration). On the other hand, if the user sees all fragmentation, allocation, and replication, the DDBMS has *no distribution transparency* and no schema integration. In the latter case, the user must refer to specific fragment copies at specific sites when formulating a query, by appending the site name before a relation or fragment name.

This is part of the complex problem of **naming** in distributed systems. In the case of a DDBMS that does not provide distribution transparency, it is up to the user to *unambiguously specify* the name of a particular relation or fragment copy. This task is more severe in a multidatabase system, because each server (local DBMS) presumably was developed independently, and as a result conflicting names may have been used at different servers. In the case of a DBMS that provides an integrated schema, however, naming becomes an internal system problem, because the user is provided with a single unambiguous schema. The DDBMS must store *all correspondences* among the integrated schema objects and the objects distributed across the various DPs in the **distribution catalog**.

23.5 Query Processing in Distributed Databases★

We now give an overview of how a DDBMS processes and optimizes a query. We first discuss the communication costs of processing a distributed query; then we discuss a special operation, called a *semijoin*, that is used in optimizing some types of queries in a DDBMS.

23.5.1 Data Transfer Costs of Distributed Query Processing

We discussed the issues involved in processing and optimizing a query in a centralized DBMS in Chapter 16. In a distributed system, several additional factors further complicate query processing. The first is the cost of transferring data over the network. This data includes intermediate files that are transferred to other sites for further processing, as well as the final result files that may have to be transferred to the site where the query result is needed. Although these costs may not be very high if the sites are connected via a high-performance local area network, they become quite significant in other types of networks. Hence, DDBMS query optimization algorithms consider the goal of reducing the *amount of data transfer* as an optimization criterion in choosing a distributed query execution strategy.

We illustrate this with two simple example queries. Suppose that the EMPLOYEE and DEPARTMENT relations of Figure 6.5 are distributed as shown in Figure 23.5. We will assume in this example that neither relation is fragmented. According to Figure 23.5, the size of the EMPLOYEE relation is $100 * 10,000 = 10^6$ bytes, and the size of the DEPARTMENT relation is $35 * 100 = 3500$ bytes. Consider the query Q: “For each employee, retrieve the employee name and the name of the department for which the employee works.” This can be stated as follows in the relational algebra:

Q: $\pi_{FNAME, LNAME, DNAME} (EMPLOYEE \bowtie_{DNO=DNUMBER} DEPARTMENT)$

The result of this query will include 10,000 records, assuming that every employee is related to a department. Suppose that each record in the query result is *40 bytes long*. The query is submitted at a distinct site 3, which is called the **result site** because the

SITE 1:

EMPLOYEE

FNAME	MINIT	LNAME	<u>SSN</u>	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
-------	-------	-------	------------	-------	---------	-----	--------	----------	-----

10,000 records

each record is 100 bytes long

SSN field is 9 bytes long

FNAME field is 15 bytes long

DNO field is 4 bytes long

LNAME field is 15 bytes long

SITE 2:

DEPARTMENT

DNAME	<u>DNUMBER</u>	MGRSSN	MGRSTARTDATE
-------	----------------	--------	--------------

100 records

each record is 35 bytes long

DNUMBER field is 4 bytes long

DNAME field is 10 bytes long

MGRSSN field is 9 bytes long

Figure 23.5 Example to illustrate volume of data transferred.

query result is needed there. Neither the EMPLOYEE nor the DEPARTMENT relations reside at site 3. There are three simple strategies for executing this distributed query:

1. Transfer both the EMPLOYEE and the DEPARTMENT relations to the result site, and perform the join at site 3. In this case we need to transfer a total of $1,000,000 + 3500 = 1,003,500$ bytes.
2. Transfer the EMPLOYEE relation to site 2, execute the join at site 2, and send the result to site 3. The size of the query result is $40 * 10,000 = 400,000$ bytes, so we must transfer $400,000 + 1,000,000 = 1,400,000$ bytes.
3. Transfer the DEPARTMENT relation to site 1, execute the join at site 1, and send the result to site 3. In this case we have to transfer $400,000 + 3500 = 403,500$ bytes.

If minimizing the amount of data transfer is our optimization criterion, we should choose strategy 3. Now consider another query Q': "For each department, retrieve the department name and the name of the department manager." This can be stated as follows in the relational algebra:

Q: $\pi_{DNAME, FNAME, LNAME} (DEPARTMENT \bowtie_{MGRSSN=SSN} EMPLOYEE)$

Again, suppose that the query is submitted at site 3. The same three strategies for executing query Q apply to Q', except that the result of Q' includes only 100 records, assuming that each department has a manager:

1. Transfer both the EMPLOYEE and the DEPARTMENT relations to the result site, and perform the join at site 3. In this case we need to transfer a total of $1,000,000 + 3500 = 1,003,500$ bytes.

2. Transfer the EMPLOYEE relation to site 2, execute the join at site 2, and send the result to site 3. The size of the query result is $40 * 100 = 4000$ bytes, so we must transfer $4000 + 1,000,000 = 1,004,000$ bytes.
3. Transfer the DEPARTMENT relation to site 1, execute the join at site 1, and send the result to site 3. In this case we have to transfer $4000 + 3500 = 7500$ bytes.

Again, we would choose strategy 3—in this case by an overwhelming margin over strategies 1 and 2. The preceding three strategies are the most obvious ones for the case where the result site (site 3) is different from all the sites that contain files involved in the query (sites 1 and 2). However, suppose that the result site is site 2; then we have two simple strategies:

4. Transfer the EMPLOYEE relation to site 2, execute the query, and present the result to the user at site 2. Here, we need to transfer the same number of bytes—1,000,000—for both Q and Q' .
5. Transfer the DEPARTMENT relation to site 1, execute the query at site 1, and send the result back to site 2. In this case we must transfer $400,000 + 3500 = 403,500$ bytes for Q and $4000 + 3500 = 7500$ bytes for Q' .

A more complex strategy, which sometimes works better than these simple strategies, uses an operation called **semijoin**. We introduce this operation and discuss distributed execution using semijoins next.

23.5.2 Distributed Query Processing Using Semijoin

The idea behind distributed query processing using the semijoin operation is to reduce the number of tuples in a relation before transferring it to another site. Intuitively, the idea is to send the *joining column* of one relation R to the site where the other relation S is located; this column is then joined with S . Following that, the join attributes, along with the attributes required in the result, are projected out and shipped back to the original site and joined with R . Hence, only the joining column of R is transferred in one direction, and a subset of S with no extraneous tuples is transferred in the other direction. If only a small fraction of the tuples in S participate in the join, this can be quite an efficient solution to minimizing data transfer.

To illustrate this, consider the following strategy for executing Q or Q' :

1. Project the join attributes of DEPARTMENT at site 2, and transfer them to site 1. For Q , we transfer $F = \pi_{\text{DNUMBER}}(\text{DEPARTMENT})$, whose size is $4 * 100 = 400$ bytes, whereas, for Q' , we transfer $F' = \pi_{\text{MGRSSN}}(\text{DEPARTMENT})$, whose size is $9 * 100 = 900$ bytes.
2. Join the transferred file with the EMPLOYEE relation at site 1, and transfer the required attributes from the resulting file to site 2. For Q , we transfer $R = \pi_{\langle \text{DNO}, \text{FNAME}, \text{LNAME} \rangle} (F \bowtie_{\text{DNUMBER}=\text{DNO}} \text{EMPLOYEE})$, whose size is $34 * 10,000 = 340,000$ bytes, whereas, for Q' , we transfer $R' = \pi_{\langle \text{MGRSSN}, \text{FNAME}, \text{LNAME} \rangle} (F' \bowtie_{\text{MGRSSN}=\text{SSN}} \text{EMPLOYEE})$, whose size is $39 * 100 = 3900$ bytes.
3. Execute the query by joining the transferred file R or R' with DEPARTMENT, and present the result to the user at site 2.

Using this strategy, we transfer 340,400 bytes for Q and 4800 bytes for Q'. We limited the EMPLOYEE attributes and tuples transmitted to site 2 in step 2 to only those that will *actually be joined* with a DEPARTMENT tuple in step 3. For query Q, this turned out to include all EMPLOYEE tuples, so little improvement was achieved. However, for Q' only 100 out of the 10,000 EMPLOYEE tuples were needed.

The semijoin operation was devised to formalize this strategy. A **semijoin** operation $R \bowtie_{A=B} S$, where A and B are domain-compatible attributes of R and S, respectively, produces the same result as the relational algebra expression $\pi_{\langle R \rangle}(R \bowtie_{A=B} S)$. In a distributed environment where R and S reside at different sites, the semijoin is typically implemented by first transferring $F = \pi_{\langle B \rangle}(S)$ to the site where R resides and then joining F with R, thus leading to the strategy discussed here.

Notice that the semijoin operation is not commutative; that is,

$$R \bowtie S \neq S \bowtie R$$

23.5.3 Query and Update Decomposition

In a DDBMS with *no distribution transparency*, the user phrases a query directly in terms of specific fragments. For example, consider another query Q: "Retrieve the names and hours per week for each employee who works on some project controlled by department 5," which is specified on the distributed database where the relations at sites 2 and 3 are shown in Figure 23.3, and those at site 1 are shown in Figure 6.6, as in our earlier example. A user who submits such a query must specify whether it references the PROJS5 and WORKS_ON5 relations at site 2 (Figure 23.3) or the PROJECT and WORKS_ON relations at site 1 (Figure 6.6). The user must also maintain consistency of replicated data items when updating a DDBMS with *no replication transparency*.

On the other hand, a DDBMS that supports *full distribution, fragmentation, and replication transparency* allows the user to specify a query or update request on the schema of Figure 6.5 just as though the DBMS were centralized. For updates, the DDBMS is responsible for maintaining *consistency among replicated items* by using one of the distributed concurrency control algorithms discussed in Section 23.6. For queries, a **query decomposition** module must break up or **decompose** a query into **subqueries** that can be executed at the individual sites. In addition, a strategy for combining the results of the subqueries to form the query result must be generated. Whenever the DDBMS determines that an item referenced in the query is replicated, it must choose or **materialize** a particular replica that is referenced during query execution.

To determine which replicas include the data items referenced in a query, the DDBMS refers to the fragmentation, replication, and distribution information stored in the DDBMS catalog. For vertical fragmentation, the attribute list for each fragment is kept in the catalog. For horizontal fragmentation, a condition, sometimes called a **guard**, is kept for each fragment. This is basically a selection condition that specifies which tuples exist in the fragment; it is called a *guard* because *only tuples that satisfy this condition* are permitted to be stored in the fragment. For mixed fragments, both the attribute list and the guard condition are kept in the catalog.

In our earlier example, the guard conditions for fragments at site 1 (Figure 6.6) are TRUE (all tuples), and the attribute lists are * (all attributes). For the fragments shown

- (a) EMPD5
 attribute list: FNAME,MINIT,LNAME,SSN,SALARY,SUPERSSN, DNO
 guard condition: DNO=5
 DEP5
 attribute list: * (all attributes DNAME,DNUMBER,MGRSSN,MGRSTARTDATE)
 guard condition: DNUMBER=5
 DEP5_LOCS
 attribute list: * (all attributes DNUMBER,LOCATION)
 guard condition: DNUMBER=5
 PROJS5
 attribute list: * (all attributes PNAME,PNUMBER,PLOCATION,DNUM)
 guard condition: DNUM=5
 WORKS_ON5
 attribute list: * (all attributes ESSN,PNO,HOURS)
 guard condition: ESSN IN (π_{SSN} (EMPD5))
 OR PNO IN ($\pi_{PNumber}$ (PROJS5))
- (b) EMPD4
 attribute list: FNAME,MINIT,LNAME,SSN,SALARY,SUPERSSN, DNO
 guard condition: DNO=4
 DEP4
 attribute list: * (all attributes DNAME,DNUMBER,MGRSSN,MGRSTARTDATE)
 guard condition: DNUMBER=4
 DEP4_LOCS
 attribute list: * (all attributes DNUMBER,LOCATION)
 guard condition: DNUMBER=4
 PROJS4
 attribute list: * (all attributes PNAME,PNUMBER,PLOCATION,DNUM)
 guard condition: DNUM=4
 WORKS_ON4
 attribute list: * (all attributes ESSN,PNO,HOURS)
 guard condition: ESSN IN (π_{SSN} (EMPD4))
 OR PNO IN ($\pi_{PNumber}$ (PROJS4))

Figure 23.6 Guard conditions and attribute lists for fragments. (a) Guards and attribute lists for relation fragments at site 2. (b) Guards and attribute lists for relation fragments at site 3.

in Figure 23.3, we have the guard conditions and attribute lists shown in Figure 23.6. When the DDBMS decomposes an update request, it can determine which fragments must be updated by examining their guard conditions. For example, a user request to insert a new EMPLOYEE tuple <'Alex', 'B', 'Coleman', '345671239', '22-APR-64', '3306 Sandstone, Houston, TX', M, 33000, '987654321', 4> would be decomposed by the DDBMS into two update requests: the first inserts the preceding tuple in the EMPLOYEE fragment at site 1, and the second inserts the projected tuple <'Alex', 'B', 'Coleman', '345671239', 33000, '987654321', 4> in the EMPD4 fragment at site 3.

For query decomposition, the DDBMS can determine which fragments may contain the required tuples by comparing the query condition with the guard conditions. For example, consider the query Q: "Retrieve the names and hours per week for each employee who works on some project controlled by department 5"; this can be specified in SQL on the schema of Figure 6.5 as follows:

Q: **SELECT** FNAME, LNAME, HOURS
 FROM EMPLOYEE, PROJECT, WORKS_ON
 WHERE DNUM=5 **AND** PNUMBER=PNO **AND** ESSN=SSN

Suppose that the query is submitted at site 2, which is where the query result will be needed. The DDBMS can determine from the guard condition on PROJS5 and WORKS_ON5 that all tuples satisfying the conditions (DNUM = 5 AND PNUMBER = PNO) reside at site 2. Hence, it may decompose the query into the following relational algebra subqueries:

T1 $\leftarrow \pi_{\text{ESSN}} (\text{PROJS5} \bowtie_{\text{PNUMBER=PNO}} \text{WORKS_ON5})$
 T2 $\leftarrow \pi_{\text{ESSN, FNAME, LNAME}} (T1 \bowtie_{\text{ESSN=SSN}} \text{EMPLOYEE})$
 RESULT $\leftarrow \pi_{\text{FNAME, LNAME, HOURS}} (T2 * \text{WORKS_ON5})$

This decomposition can be used to execute the query by using a semijoin strategy. The DDBMS knows from the guard conditions that PROJS5 contains exactly those tuples satisfying (DNUM = 5) and that WORKS_ON5 contains all tuples to be joined with PROJS5; hence, subquery T1 can be executed at site 2, and the projected column ESSN can be sent to site 1. Subquery T2 can then be executed at site 1, and the result can be sent back to site 2, where the final query result is calculated and displayed to the user. An alternative strategy would be to send the query Q itself to site 1, which includes all the database tuples, where it would be executed locally and from which the result would be sent back to site 2. The query optimizer would estimate the costs of both strategies and would choose the one with the lower cost estimate.

23.6 Overview of Concurrency Control and Recovery in Distributed Databases★

For concurrency control and recovery purposes, numerous problems arise in a distributed DBMS environment that are not encountered in a centralized DBMS environment. These problems include the following:

- *Dealing with multiple copies of the data items:* The concurrency control method is responsible for maintaining consistency among these copies. The recovery method is responsible for making a copy consistent with other copies if the site on which the copy is stored fails and recovers later.
- *Failure of individual sites:* The DDBMS should continue to operate with its running sites, if possible, when one or more individual sites fail. When a site recovers, its local database must be brought up to date with the rest of the sites before it rejoins the system.
- *Failure of communication links:* The system must be able to deal with failure of one or more of the communication links that connect the sites. An extreme case of this problem is that **network partitioning** may occur. This breaks up the sites into two or more partitions, where the sites within each partition can communicate only with one another and not with sites in other partitions.

- *Distributed commit*: Problems can arise with committing a transaction that is accessing databases stored on multiple sites if some sites fail during the commit process. The **two-phase commit protocol** (see Section 19.5) is often used to deal with this problem.
- *Distributed deadlock*: Deadlock may occur among several sites, so techniques for dealing with deadlocks must be extended to take this into account.

Distributed concurrency control and recovery techniques must deal with these and other problems. In the following subsections, we review some of the techniques that have been suggested to deal with recovery and concurrency control in DDBMSs.

23.6.1 *Distributed Concurrency Control Based on a Distinguished Copy of a Data Item*

To deal with replicated data items in a distributed database, a number of concurrency control methods have been proposed that extend the concurrency control techniques for centralized databases. We discuss these techniques in the context of extending centralized *locking*. Similar extensions apply to other concurrency control techniques. The idea is to designate a *particular copy* of each data item as a **distinguished copy**. The locks for this data item are associated *with the distinguished copy*, and all locking and unlocking requests are sent to the site that contains that copy.

A number of different methods are based on this idea, but they differ in their method of choosing the distinguished copies. In the **primary site** technique, all distinguished copies are kept at the same site. A modification of this approach is the primary site with a **backup site**. Another approach is the **primary copy** method, where the distinguished copies of the various data items can be stored in different sites. A site that includes a distinguished copy of a data item basically acts as the **coordinator site** for concurrency control on that item. We discuss these techniques next.

Primary Site Technique. In this method a single **primary site** is designated to be the **coordinator site for all database items**. Hence, all locks are kept at that site, and all requests for locking or unlocking are sent to that site. This method is thus an extension of the centralized locking approach. For example, if all transactions follow the two-phase locking protocol, serializability is guaranteed. The advantage of this approach is that it is a simple extension of the centralized approach and hence is not overly complex. However, it has certain inherent disadvantages. One disadvantage is that all locking requests are sent to a single site, possibly overloading that site and causing a system bottleneck. Another disadvantage is that failure of the primary site paralyzes the system, since all locking information is kept at that site. This can limit system reliability and availability.

Although all locks are accessed at the primary site, the items themselves can be accessed at any site at which they reside. For example, once a transaction obtains a `read_lock` on a data item from the primary site, it can access any copy of that data item. However, once a transaction obtains a `write_lock` and updates a data item, the DDBMS is responsible for updating *all copies* of the data item before releasing the lock.

Primary Site with Backup Site. This approach addresses the second disadvantage of the primary site method by designating a second site to be a **backup site**. All locking information is maintained at both the primary and the backup sites. In case of failure of the primary site, the backup site can take over as primary site, and a new backup site can be chosen. This simplifies the process of recovery from failure of the primary site, since the backup site takes over and processing can resume after a new backup site is chosen and the lock status information is copied to that site. It slows down the process of acquiring locks, however, because all lock requests and granting of locks must be recorded at *both the primary and the backup sites* before a response is sent to the requesting transaction. The problem of the primary and backup sites becoming overloaded with requests and slowing down the system remains undiminished.

Primary Copy Technique. This method attempts to distribute the load of lock coordination among various sites by having the distinguished copies of different data items *stored at different sites*. Failure of one site affects any transactions that are accessing locks on items whose primary copies reside at that site, but other transactions are not affected. This method can also use backup sites to enhance reliability and availability.

Choosing a New Coordinator Site in Case of Failure. Whenever a coordinator site fails in any of the preceding techniques, the sites that are still running must choose a new coordinator. In the case of the primary site approach with no backup site, all executing transactions must be aborted and restarted, and the recovery process is quite tedious. Part of the recovery process involves choosing a new primary site and creating a lock manager process and a record of all lock information at that site. For methods that use backup sites, transaction processing is suspended while the backup site is designated as the new primary site and a new backup site is chosen and is sent copies of all the locking information from the new primary site.

If a backup site X is about to become the new primary site, X can choose the new backup site from among the system's running sites. However, if no backup site existed, or if both the primary and the backup sites are down, a process called **election** can be used to choose the new coordinator site. In this process, any site Y that attempts to communicate with the coordinator site repeatedly and fails to do so can assume that the coordinator is down and can start the election process by sending a message to all running sites proposing that Y become the new coordinator. As soon as Y receives a majority of yes votes, Y can declare that it is the new coordinator. The election algorithm itself is quite complex, but this is the main idea behind the election method. The algorithm also resolves any attempt by two or more sites to become coordinator at the same time. The references in the bibliography at the end of this chapter discuss the process in detail.

23.6.2 Distributed Concurrency Control Based on Voting

The concurrency control methods for replicated items discussed earlier all use the idea of a distinguished copy that maintains the locks for that item. In the **voting method**, there is no distinguished copy; rather, a lock request is sent to all sites that include a copy of the data item. Each copy maintains its own lock and can grant or deny the

request for it. If a transaction that requests a lock is granted that lock by a *majority* of the copies, it holds the lock and informs *all copies* that it has been granted the lock. If a transaction does not receive a majority of votes granting it a lock within a certain *time-out period*, it cancels its request and informs all sites of the cancellation.

The voting method is considered a truly distributed concurrency control method, since the responsibility for a decision resides with all the sites involved. Simulation studies have shown that voting has higher message traffic among sites than do the distinguished copy methods. If the algorithm takes into account possible site failures during the voting process, it becomes extremely complex.

23.6.3 Distributed Recovery

The recovery process in distributed databases is quite involved. We give only a very brief idea of some of the issues here. In some cases it is quite difficult even to determine whether a site is down without exchanging numerous messages with other sites. For example, suppose that site X sends a message to site Y and expects a response from Y but does not receive it. There are several possible explanations:

- The message was not delivered to Y because of communication failure.
- Site Y is down and could not respond.
- Site Y is running and sent a response, but the response was not delivered.

Without additional information or the sending of additional messages, it is difficult to determine what actually happened.

Another problem with distributed recovery is distributed commit. When a transaction is updating data at several sites, it cannot commit until it is sure that the effect of the transaction on *every* site cannot be lost. This means that every site must first have recorded the local effects of the transactions permanently in the local site log on disk. The two-phase commit protocol, discussed in Section 19.5, is often used to ensure the correctness of distributed commit.

23.7 Summary

In this chapter we provided an introduction to distributed databases. This is a very broad topic, and we discussed only some of the basic techniques used with distributed databases. We first discussed the reasons for distribution and the potential advantages of distributed databases over centralized systems. We then described the client-server architecture of a DDBMS. We defined the concepts of data fragmentation, replication, and distribution, and we distinguished between horizontal and vertical fragments of relations. We discussed the use of data replication to improve system reliability and availability. We also defined the concept of distribution transparency and the related concepts of fragmentation transparency and replication transparency. We categorized DDBMSs by using criteria such as degree of homogeneity of software modules and degree of local autonomy.

In Section 23.5, we illustrated some of the techniques used in distributed query processing. The cost of communication among sites is considered a major factor in dis-

tributed query optimization. We compared different techniques for executing joins and presented the semijoin technique for joining relations that reside on different sites.

In Section 23.6, we briefly discussed the concurrency control and recovery techniques used in DDBMSs. We reviewed some of the additional problems that must be dealt with in a distributed environment that do not appear in a centralized environment.

Review Questions

- 23.1. What are the main reasons for and potential advantages of distributed databases?
- 23.2. What additional functions does a DDBMS have over a centralized DBMS?
- 23.3. What are the main software modules of a DDBMS? Discuss the main functions of each of these modules in the context of the client-server architecture.
- 23.4. What is a fragment of a relation? What are the main types of fragments? Why is fragmentation a useful concept in distributed database design?
- 23.5. Why is data replication useful in DDBMSs? What typical units of data are replicated?
- 23.6. What is meant by *data allocation* in distributed database design? What typical units of data are distributed over sites?
- 23.7. How is a horizontal partitioning of a relation specified? How can a relation be put back together from a complete horizontal partitioning?
- 23.8. How is a vertical partitioning of a relation specified? How can a relation be put back together from a complete vertical partitioning?
- 23.9. Discuss what is meant by the following terms: *degree of homogeneity of a DDBMS*, *degree of local autonomy of a DDBMS*, *federated DBMS*, *distribution transparency*, *fragmentation transparency*, *replication transparency*, *multidatabase system*.
- 23.10. Discuss the naming problem in distributed databases.
- 23.11. Discuss the different techniques for executing an equijoin of two files located at different sites. What main factors affect the cost of data transfer?
- 23.12. Discuss the semijoin method for executing an equijoin of two files located at different sites. Under what conditions is an equijoin strategy efficient?
- 23.13. Discuss the factors that affect query decomposition. How are guard conditions and attribute lists of fragments used during the query decomposition process?
- 23.14. How is the decomposition of an update request different from the decomposition of a query? How are guard conditions and attribute lists of fragments used during the decomposition of an update request?
- 23.15. Discuss the factors that do not appear in centralized systems that affect concurrency control and recovery in distributed systems.
- 23.16. Compare the primary site method with the primary copy method for distributed concurrency control. How does the use of backup sites affect each?
- 23.17. When are voting and elections used in distributed databases?

Exercises

- 23.18. Consider the data distribution of the COMPANY database, where the fragments at sites 2 and 3 are as shown in Figure 23.3 and the fragments at site 1 are as shown in Figure 6.6. For each of the following queries, show at least two strategies of decomposing and executing the query. Under what conditions would each of your strategies work well?
- For each employee in department 5, retrieve the employee name and the names of the employee's dependents.
 - Print the names of all employees who work in department 5 but who work on some project *not* controlled by department 5.

Selected Bibliography

The textbooks by Ceri and Pelagatti (1984a) and Ozsu and Valduriez (1990) are devoted to distributed databases. Fragmentation is discussed in Chang and Cheng (1980) and Ceri and Pelagatti (1984a). Federated database systems are discussed in McLeod and Heimbigner (1985). Principles of computer communication networks are discussed in the textbook by Tanenbaum (1981).

Distributed query processing, optimization, and decomposition are discussed in the papers by Hevner and Yao (1979), Apers et al. (1983), Ceri and Pelagatti (1984), Bodorick et al. (1992), and Kerschberg et al. (1982). Bernstein and Goodman (1981) discusses the theory behind semijoin processing. Wong (1983) discusses the use of relationships in relation fragmentation. Concurrency control and recovery schemes are discussed and compared in Garcia-Molina (1978), Ries (1979), and Bernstein and Goodman (1981a). Elections in distributed systems are discussed in Garcia-Molina (1982).

A concurrency control technique for replicated data that is based on voting is presented by Thomas (1979). Gifford (1979) proposes the use of weighted voting, and Paris (1986) describes a method called voting with witnesses. Jajodia and Mutchler (1990) discusses dynamic voting. A technique called *available copy* is proposed by Bernstein and Goodman (1984), and one that uses the idea of a group is presented in ElAbbadi and Toueg (1988). Other recent work that discusses replicated data includes Gladney (1989), Agrawal and ElAbbadi (1990), ElAbbadi and Toueg (1990), Kumar and Segev (1993), Mukkamala (1989), and Wolfson and Milo (1991). Schlageter (1981), Bassiouni (1988), and Ceri and Owicki (1983) discuss optimistic methods for DDB concurrency control. Garcia-Molina (1983) and Kumar and Stonebraker (1987) discuss a technique that uses the semantics of the transactions. Distributed concurrency control techniques based on locking and distinguished copies are presented by Menasce et al. (1980) and Minoura and Wiederhold (1982). Obermark (1982) presents algorithms for distributed deadlock detection. Lamport (1978) discusses problems with generating unique timestamps in a distributed system.

A survey of recovery techniques in distributed systems is given by Kohler (1981). Skeen (1981) discusses nonblocking commit protocols, and Reed (1983) discusses atomic actions on distributed data. A book edited by Bhargava (1987) presents various approaches and techniques for distributed recovery and concurrency control.

Techniques for schema integration in federated databases are presented by Elmasri et al. (1986), Hayne and Ram (1990), and Motro (1987). Onuegbue et al. (1983) discusses the translation and local optimization problem. Elmagarmid and Helal (1988) and Gamal-Eldin et al. (1988) discuss the update problem in heterogeneous DDBSs. Heterogeneous distributed databases are also discussed in Kaul et al. (1990), Hsiao and Kamel (1989), and Perrizo et al. (1991).

Techniques for distributed data allocation are proposed in Blankenship et al. (1991), Morgan and Levin (1977), Ramamoorthy and Wah (1979), Chu and Hurley (1982), and Kamel and King (1985), among others. Distributed database design involving horizontal and vertical fragmentation, allocation, and replication is addressed in Ceri et al. (1983), Navathe et al. (1984), Ceri et al. (1982), Wilson and Navathe (1986), and Elmasri et al. (1987).

Recently, multidatabase systems and interoperability have become important topics. Techniques for dealing with semantic incompatibilities among multiple databases are examined in DeMichiel (1989), Siegel and Madnick (1991), Krishnamurthy et al. (1991), and Wang and Madnick (1989). Transaction processing in multidatabases is discussed in Mehrotra et al. (1992), Georgakopoulos et al. (1991), Elmagarmid et al. (1990), and Brietbart et al. (1990), among others.

A number of experimental distributed DBMSs have been implemented. These include distributed INGRES (Epstein et al. 1978), DDTS (Devor and Weeldreyer 1980), SDD-1 (Rothnie et al. 1980), System R* (Lindsay et al. 1984), SIRIUS-DELTA (Ferrier and Stangret 1982), and MULTIBASE (Smith et al. 1981). The OMNIBASE system (Rusinkiewicz et al. 1988) is a federated DDBMS. Many commercial DBMS vendors have announced distributed versions of their systems. Commercial DDBMSs are now applying the client-server architecture in their newer versions. Some issues concerning client-server systems are discussed in Carey et al. (1991), DeWitt et al. (1990), and Wang and Rowe (1991).

Emerging Database Technologies and Applications

So far in this book we have discussed the various techniques for modeling data, designing databases, and implementing them on computers. The term **database technology** is used to refer collectively to the *proven techniques* used on a *wide scale* in industry and government, as well as by individual personal database users, to perform these functions on a *day-to-day* basis. The bulk of what we have discussed is already a part of current database technology. In every discipline of human endeavor, technology always lags behind ongoing research by several years. Database technology is no exception. Therefore, the research work underway today that has the potential of becoming a viable technology will not be available for general consumption for another 5 to 10 years. In this textbook it is impossible for us to include details of a number of such ongoing promising research efforts. We intend, however, to give the reader a broad perspective on the current state of the art in databases and database systems research in this chapter. We will attempt to bring out the major issues and to point out the difficult problems, rather than to discuss the proposed solutions in detail. References to pertinent literature are given whenever appropriate. A few additional references appear in the bibliography.

This chapter is organized as follows: Section 25.1 is an overview of the progression of database technology, roughly over the last 30 years. Section 25.2 discusses the emerging nonconventional areas of applications that are making use of this technology and will continue to place further demands on it. Section 25.3 describes a number of new technologies that are expected to be incorporated in future DBMSs. We end the chapter with Section 25.4, where we outline some emerging technologies and give pointers to research needed in the near future.

One **disclaimer**: This chapter does *not* provide a complete summary of ongoing research. Omissions of some research areas should not be considered to imply any bias. The vast field of database management research, where every year at least 200 to 300

papers are published through refereed conferences and journals, is difficult to condense in this small chapter.

25.1 Progression of Database Technology

Table 25.1 summarizes the progression of database technology for the last three decades or so. The table's first three columns correspond roughly to the three identifiable periods. The last column lists expected future developments. We discuss this table row by row.

25.1.1 Data Models

The network and hierarchical models emerged in the 1960s. Since its introduction in 1970, the relational model has excited rapidly increasing interest because of its desirable properties, including its formal basis, homogeneity, well-defined set of algebraic operations, and calculus-based languages. Shortcomings of the relational model in terms of its semantic expressive power led to interest in semantic models. This has grown since the late 1970s, particularly with the advent of the Entity-Relationship (ER) model (Chen 1976), Semantic Hierarchy Model (Smith and Smith 1977), and Semantic Data Model (Hammer and McLeod 1981). In his RM/T model, Codd (1979) suggested that there is a need to add more expressive power to the relational data model by providing it with certain abstractions from the preceding models. The interest in semantic models continued in the 1980s. And today, with the expanding horizons of database applications (Section 25.2), the need for the abstractions that we discussed in Chapter 21 is being felt even more intensely.

A major trend in the next several years will be object-oriented (OO) data models and object-oriented database management systems (OODBMSs); we discussed both of these in detail in Chapter 22. OO data models bring an expressive power similar to that of semantic data models, but go beyond them by explicitly modeling behavior. They are claimed to be easier to implement and use for application development due to the built-in modularity, encapsulation and reuse of code.

Logic is being proposed as an underlying data model for relational and other representation schemas. The relational calculus (see Chapter 8) is based on a branch of logic called first-order predicate calculus; hence, use of logic already exists to characterize relational queries. The relationship between logic and relational databases is well brought out in Gallaire et al. (1984). Logic provides a formalism that can be used in query languages, integrity modeling, query evaluation, treatment of null values, dealing with incomplete information, and so on. Logic also leads to a formal understanding of deduction in databases, which we discussed in Chapter 24.

The future is likely to see further proliferation of object-oriented DBMSs, further use of logic for deductive databases, and a merging of knowledge representation, programming languages, and data models (Brodie et al. 1984; Bancilhon and Buneman 1990; Atkinson and Buneman 1987).

Another distinct trend in commercial DBMS development involves producing unified systems with "hybrid" data models. One example is the UniSQL system which supports a full object-oriented data model that is completely compatible with the relational

Table 25.1 Database Technology Trends

	1960s to Mid-1970s	1970s to Mid-1980s	1980s to Early 1990s	Future
<i>Data Model</i>	Network Hierarchical	Relational	Semantic Object-oriented Logic Faster PCs Workstations Database machines Back ends Graphics Menus Query-by-forms	Merging data models with knowledge representation Hybrid models Client-server configuration Parallel processing Optical memories
<i>Database Hardware</i>	Mainframes	Mainframes Minis PCs		
<i>User Interface</i>	None	Query languages Forms		Multimedia Natural languages Speech input Freehand text Integrated database and programming languages
<i>Program Interface</i>	Procedural	Embedded query languages	Standardized SQL 4GL Logic programming	
<i>Presentation and Display</i>	Reports	Report generators	Business graphics Image output	Generalized presentation managers Distributed, heterogeneous data and knowledge processing with multimedia information Parallel database management
<i>Processing</i>	Processing data	Information and transaction processing	Transaction processing Knowledge processing	

model. The open ODB product of Hewlett-Packard, based on the IRIS DBMS developed earlier, combines an object-oriented data model with a functional data model. Vendors of conventional DBMS products have already begun implementing a relational SQL-based interface to allow concurrent development of relational applications. This has happened with network DBMSs such as UNISYS's DMS1100 and Computer Associates' IDMS, and it will continue as a trend.

25.1.2 Database Hardware

Over the last 30 years there has been quite a revolution in growth of storage capacities of computers, microminiaturization of the circuitry, and increased processing speeds. The cost of components has fallen steadily and dramatically. The amount of data that can be processed in a certain amount of time has been growing. Moreover, larger databases are becoming feasible on smaller equipment. Until a few years ago, sizable databases could only be handled by mainframes and minis. With the advent of powerful hardware, it is now possible to implement major DBMS products on workstations and personal computers. Relational DBMSs such as DB2/2, ORACLE, and INFORMIX and object-oriented DBMSs such as ObjectStore and O2 (which we discussed in Chapter 22) are available on these. The functionality of low-end database products such as Lotus 1-2-3, the DBase series, ACCESS, and Paradox is also continuously expanding with more powerful hardware that affords windowing and graphics capabilities as well as higher speeds.

Even though storage costs have declined and main storage capacities of systems have grown considerably, the overall performance of database systems is still a great concern. In addition, the advances in software, which incorporate sophisticated data modeling, have been very gradual (as we shall see in Section 25.2) compared to those in processors and I/O architecture. This is due to the mismatch between access speeds of secondary storage devices and those of central processing units, leading to inevitable delays in processing large volumes of data stored on disk. Compared to the advances in speeds of processors (attributed to new logic technologies as well as to the availability of parallel processing), the rate of increase in speeds of secondary storage devices (mainly disks) has been slow.

To deal with these problems, several alternative suggestions have been made for special hardware geared for the data management functions. These alternatives, generically known as **database machines** or computers, include back-end processors, intelligent (logic per track) devices, multiprocessor systems, associative memory systems, and special-purpose processors. The Intelligent Database Machine (IDM) introduced by Britton-Lee in 1982 is the first commercial database machine. Teradata's DBC/1012 is a recent entry; it is the only multiprocessor database computer available commercially that has a very flexible reconfigurable architecture. Another database machine called the Teradata DBC/1012 has been introduced. A parallel database management system implementation of the ORACLE relational DBMS exists on the KSR (Kendall Square Research) parallel processing hardware, as well as on the Ncube processor. Research on alternative architectures and simulation of their performance are ongoing activities. Su (1988) provides an excellent survey of research efforts in database computers. Typically, however, most of them have remained as designs on paper or small prototype implementations because of lack of resources to build actual full-scale hardware or even prototypes.

Because of the demands of high-volume transaction processing and of ever-increasing processor speeds (especially in supercomputers), much attention will be given in the future to database machines and to parallel processing architectures. From the experience of the last 15 years, however, it is hard to predict what, if anything, will emerge in the way of economically viable and commercially feasible solutions.

From the standpoint of new storage hardware, optical storage devices (which are currently write-once and read-forever types) are being modified into read-write devices and will soon be economically affordable. They have very high capacities (in the range of 10^{12} bytes) and open up the possibility of storing vast amounts of information by recording updated information without erasing old information. Read-write optical disks have already been introduced on the market. Even with the moderate-cost write-once optical disks available today, which have a physical write rate of one page per second, a disk used for a normal workload can last one year (Gait 1988). This opens up a number of exciting new prospects for historical data storage and retrieval (see temporal applications in Section 25.3.4).

25.1.3 User Interfaces

In Chapter 1 we pointed out the large spectrum of people who are typical users of DBMSs. When DBMSs were introduced in the late 1960s, end users had no direct access to databases. Their interactions were limited to verbally communicating their requirements to programmers. Filling forms was one of the first interactive modes of data entry suitable for a large population of end users such as sales or insurance clerks. Most interaction still occurred with batch programs in procedural languages.

Query languages became popular in the 1970s. We discussed the query languages SQL (Chapter 7) and QBE and QUEL (Chapter 8) for the relational data model. It is quite remarkable that, for all practical purposes, such a major DBMS product as IMS (see Chapter 11 for details) does *not* have a query language. The network data model has also survived without much query language support except for specialized high-level user commands that customers and vendors may have developed on their own. The availability of graphics workstations and windowing has tremendously increased the potential for creating nonsyntactic interfaces to DBMSs, where a user is not burdened with having to use a specific syntax. Interfaces making extensive use of windowing capabilities, pull-down menus, and icons are likely to grow and become more sophisticated.

Most DBMSs are aiming to add facilities called **Query by Forms**. These allow users to invoke “parameterized” queries for which run-time parameters are supplied and a pre-defined query is recompiled or reinterpreted and executed. Other user facilities include *icon-based* interfaces, where a user touches icons or images on the screen to formulate a query, and *mouse-based* interfaces, where a *mouse* or *touch-sensitive* screen permits cursor movement while specifying a query. Some interfaces access actual data and provide values existing in the database in a scrollable window so that these values can be used as constants in queries.

Natural language interfaces have been explored for some time. Three basic approaches are used to interpret natural language:

- *Keyword extraction or pattern matching*: In keyword systems, the program relates words in natural language to specific fields in a database, where an application

developer defines the relationships. However, pattern matching without a grammatical basis is of limited use.

- *Parsing*: Parsing converts a potentially ambiguous phrase in English to an internal format that should accurately represent the user's intended query. Two variations on this approach are possible: one based on a grammar of the natural language, and another based on the semantics of the language in terms of a lexicon and a series of production rules. The LADDER system (Hendrix 1978) used a semantic grammar for accessing U.S. Navy databases. Harris's (1978) INTELLECT system, now a commercial product, is also based on grammar rules. Codd's (1978) Rendezvous system used a lexicon and converted natural language queries into relational calculus based on a phrase grammar.
- *Query mapping*: A knowledge base or a "world model" is used with canonical representations of sentences from a certain application domain, together with linguistic knowledge and mapping knowledge, to map natural language queries at a conceptual level into database queries in a specific language. The KID system (Ishikawa 1987) is an example of this approach.

Today's natural language systems suffer from the fact that they represent a heavy overhead, require a database-specific lexicon to be built, and allow ambiguities in query specification. As a result, the user may not be aware of how the system has actually interpreted a query and the system has no way of predicting exactly what the user may want. Icon- and mouse-based interfaces are equally easy to operate at times and are much more economical.

The technology of *speech recognition and understanding* is still not sophisticated enough to allow speech input to be viable in a major way in the near future. That approach, however, has a phenomenal potential for opening the doors of databases to people of all ages and all levels of sophistication and for making database-based computer-aided instruction available to the masses.

25.1.4 Program Interfaces

Even today, the bulk of programming in the commercial world, which may account for 70 to 80% (a rough guess!) of actual database processing, is done in conventional programming languages—mostly in COBOL, PL/1, and (now) C. These languages, plus FORTRAN, Assembler, and ALGOL, have been the mainstay of database applications. The C++ language has recently entered the scene and is gaining ground rapidly, especially in object-oriented DBMSs.

The 1970s saw the emergence of query languages that were embedded in programming languages for large-scale application program development. We saw examples of *embedding SQL* in Chapter 7.

A departure from this trend came in two areas. First, the **Fourth-Generation Languages** (4GLs, a term coined commercially without a precise definition) became popular in the 1980s. They allow the user to give a high-level specification of an application in the 4GL language. The system (or a tool) then *automatically generates* the application code. For example, INGRES and most RDBMSs provide an Application by Forms interface, where the application designer develops an application by using forms displayed on the

screen rather than by writing a program for it. 4GLs, however, have not been uniformly accepted for general-purpose data management and for most report generation. Their built-in database support is minimal.

Another major development in the 1980s has been the standardization of SQL, first the ANSI SQL and SQL 89 standards, and more recently SQL2 and SQL3. Chapter 7 follows SQL2 syntax. Most RDBMS vendors are trying to comply with these standards, which will facilitate interoperability among these systems. SQL3 is in a draft form at the time of our publication; it combines many object-oriented features (see Chapter 22) in SQL. Many nonrelational DBMS vendors have opted to provide SQL programming interfaces for their DBMS—for example, in the IDMS network model-based system (see Section 10.7) and in the ADABAS system of software AG.

Use of logic programming languages for deductive database management* (see Chapter 24) has not had much impact to date. One reason for this is the lack of full-scale commercial systems. We discussed the only such system, LDL, in Section 24.7. We expect that the trend will be toward *integrated programming systems*, where the inferencing and recursion of the logic programming language are coupled with the efficient manipulation of facts in the DBMS. With the merger of object-oriented and relational data models, or with a relational interface developed on top of a network DBMS, future systems will continue to present multiple programming interface options within a single system for application development.

25.1.5 Presentation and Display

The original output devices on computer systems were line printers and card punchers. Most output was presented either as line-by-line reports or as calculated values punched into cards and subjected to further processing. The language RPG (Report Generation Language) was the beginning of a trend toward making report generation a specialized activity that could be specified in terms of high-level report generation commands.

Today, report generators are a standard feature of most DBMSs. Examples are INGRES's report writer, IDMS's CULPRIT package, and UNIFY's lst and rpt processors. Special report definition languages are provided in which users specify formatting, spacing, pagination, levels of totals, headers, justification, and so on.

With interactive access to databases for a variety of users, report generation became possible by invoking the report generator or by specifying reports via forms (for example, INGRES's Report by Forms). With the advent of PCs, users have come to expect to be presented with a great deal of information on the screen. Most DBMS packages supply **business graphics** output in a variety of forms, including xy plots (scatter plots) using linear regression, bar charts, pie charts, and line plots (containing piecewise linear approximations of curves that appear as a set of connected lines). With color graphics, aesthetically pleasing as well as impressive displays are now commonplace. These types of displays require the query output to be postprocessed by a display manager.

*We do not intend to define deductive databases formally here. Suffice it to say that a conventional database becomes deductive by the addition of a theory with certain axioms and deductive laws. See Gallaire et al. (1984).

By integrating image technology with database processing and by storing digitized voice as data, some of today's experimental systems are able to produce image and voice output. Digitized image storage is becoming quite common. Specialized hardware for speech synthesis may become commercially available in the next few years.

25.1.6 Nature of Processing

Database technology was originally introduced as a response to the problems of file processing and its associated disadvantages (see Section 1.3). The main applications are in business data processing—inventory management, payroll, general ledger billing, order processing, sales reporting, and so on.

During the late 1970s, the technology began to be applied in a variety of domains from different disciplines. High-level applications that summarized data into information for tactical and strategic planning were generated. Reliable and secure DBMSs became a resource of centralized information with which a large number of geographically separated applications/users could interact. This has been made possible by advances in **data communications technology**, including broadband and satellite communication networks. The major users of DBMS technology today use it for **transaction processing**.

Airlines, major hotel chains, insurance companies, banks, retail chain stores, automobile and appliance dealerships, and the like operate today by using reliable rapid transaction processing and on-line real-time updates. Today it is possible to book a seat in a theatre on Broadway from a remote town in Oregon. These applications will continue to place higher and higher demands on the technology of on-line transaction processing.

25.1.7 Current Trends in Technology

The trend of technology for the next several years will be dominated by the following themes:

- *Distributed, heterogeneous environments*: Large centralized databases will be replaced by distributed databases. These databases may be implemented from conventional network and hierarchical DBMSs, the current relational DBMSs, or the object-oriented DBMSs that are gaining popularity. The “client-server” architecture is becoming extremely popular and will afford users various options with regard to the functionality of the DBMS and the location of data.
- *Open systems*: Several standardization efforts such as SQL Access Group, ANSI/X3H7, Object Management Group's (OMG) Object Data Management Group (ODMG), Microsoft's ODBS, IDAPI, and ODAPI (Borland) are striving to improve the “openness” of future database system and applications. Systems that comply with these standards will be able to communicate among themselves easily. SQL2 and SQL3 standards will play a major role here. The trend will be toward designing global applications that run on clients' systems and draw data from a variety of servers with standardized data access protocols.
- *More functionality*: The sheer functionality of DBMSs will go beyond the conventional define, store, access, query, and report functions. The deductive capability described in Chapter 24, the processing of meta-data, and better specification of

behavior through methods (Chapter 22) will be included. Additionally, the “active” capability (which we describe in Section 25.3) to make databases more responsive, multimedia database management, better processing of temporal and spatial data, special DBMSs for scientific users, and so on are likely to dominate the development of database technology in coming years. We address a few of these areas and their special needs in Section 25.3.

- *Parallel Database Management:* A new breed of database management systems is likely to be offered on a commercial scale based on the MIMD (multiple instruction, multiple data paths) architectures such as Intel’s iPSC hypercube or the Kendall Square Research machine. Current work involves new algorithms for parallel query strategies for data allocation and effective use of the overall system to minimize the cost per transaction. Hardly any work has thus far been done on how an existing array of database applications should be parallelized to benefit from the parallel DBMSs of the future.

25.2 Emerging Database Applications

The applications of database technology are expanding continually. Since data is at the heart of any information system, as computing becomes more widely available, each discipline will come up with its own unique set of applications. It is not our objective to go through such lists of applications in detail. We wish to identify major categories of applications that are recognized today as presenting good potential as well as a great challenge to database technology. Special modeling characteristics of each application area will be highlighted.

25.2.1 Engineering Design and Manufacturing

The difficult goal of computer-integrated design and manufacturing requires the effective management of design and manufacturing information. This topic spans subareas denoted by several acronyms: CAD (computer-aided design), CAM (computer-aided manufacturing), CAE (computer-aided engineering), and CIM (computer-integrated manufacturing). An integrated approach to managing manufacturing information requires compatible representation and manipulation of information in different phases of the product life cycle. This includes business applications such as sales forecasting, order processing, product planning, production control, inventory control, and cost accounting; design and engineering of the product together with planning for materials requirements; manufacturing-related applications, including manufacturing resource planning and control; and applications of group technology for parts classification, robotics, and manufacturing process control (including NC—numerical control programming) (Bray 1988).

The standard advantages of database technology help to integrate these areas (see Sections 1.6 and 1.7). A great deal of research is being directed toward both the design and the manufacturing problems, and a major emphasis today is placed on developing DBMSs particularly suited to computer-aided design applications. Figure 25.1 shows the variety of data that is relevant and how a number of applications may be supported in the design environment once this data is captured.

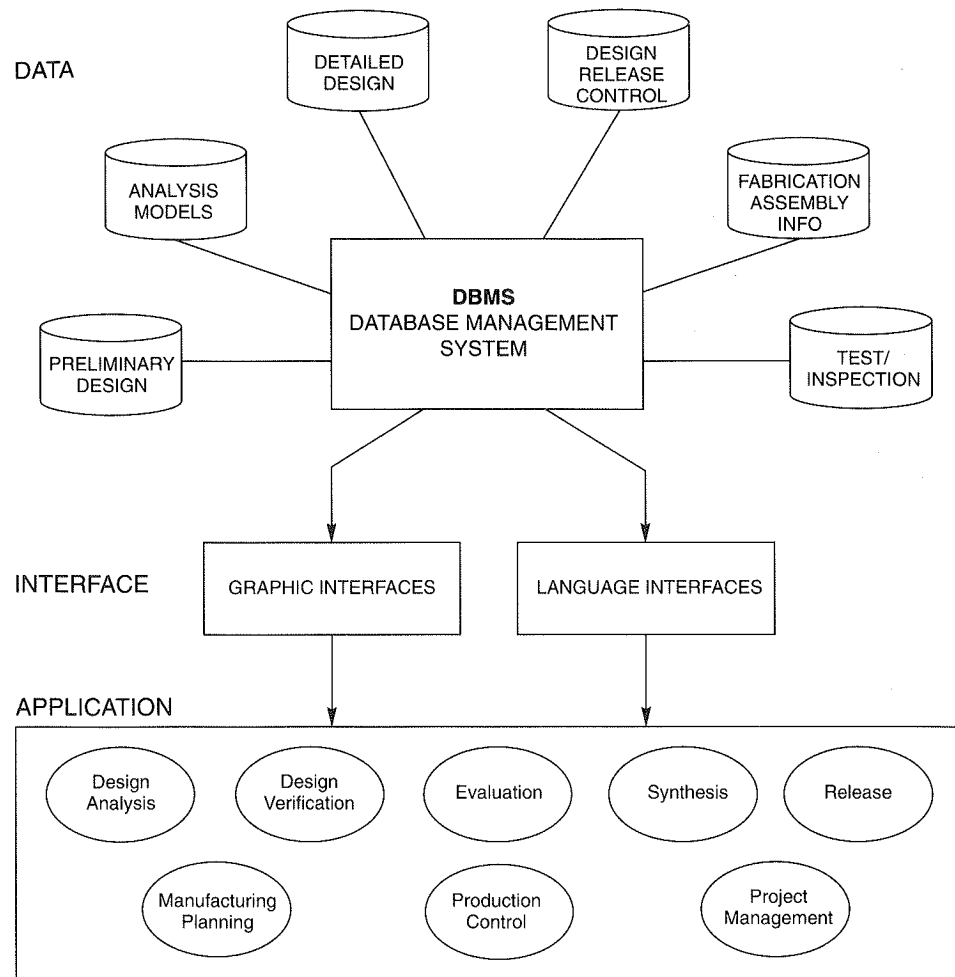


Figure 25.1 Use of a database management system in engineering design and applications.

Engineering design is an exploratory and iterative process. The engineering design activity for complex systems such as airplanes or automobiles is conducted by project teams, and the design of a component or a subsystem continuously evolves under a set of design guidelines, resource limitations, and design constraints. Intermittently, designs are cross-checked against other independently evolving designs, and finally, “permanent” designs or versions are stored.

Role of Database Management in CAD. Two important areas of investigation are the design of VLSI (very large-scale integration) electronic systems and the design of mechanical structures and systems. A basic challenge in mechanical design is **geometric modeling**, which refers to representing the physical shape of mechanical parts. The shape of

parts provides a common thread through the design, analysis, and manufacturing cycle for parts. Different CAD systems employ different geometric modeling techniques, including wire frame modeling, surface modeling, and solid modeling. It is a traumatic experience for the user to transport data among different CAD systems or from a CAD system to a CAM system. The National Bureau of Standards defined a common neutral interface or format of data called Initial Graphics Exchange Specification (IGES 1983) to facilitate such interchange of data. IGES defines standard design entities for 2D drawings and 3D objects as well as electrical diagrams. For three-dimensional solids, two types of representations, called boundary representation and constructive solid geometry, are more common.

Structural engineering deals with problems ranging from the design of trusses in buildings to the design of complex launch pad assemblies for spacecraft. Whether buildings, mechanical assemblies, bridges, or spaceships are involved, structures are constrained by geometries and require the selection of an optimal set of members to meet weight constraints and design limits and provide reliability under different loadings. Databases can play an important role in storing information centrally for use with analytical tools (such as finite-element analysis), graphics (typically three-dimensional graphics on CAD systems), simulations, and optimal design algorithms. In chemical engineering, databases are used for plant design and process control. There are two main reasons for using a centralized database system for design data:

- Part of a design can be synthesized, analyzed, coordinated, and documented while individual project teams work on different parts of the design.
- Constraints related to identical standards, designs, and specifications and other physical properties, design style, and topological relationships can be verified and enforced automatically.

Engineering design data is difficult to capture and represent with conventional data models (Chapters 6–11) and DBMSs for the following reasons:

- Engineering design data contains a nonhomogeneous collection of design objects. Classical data models deal with homogeneous collections.
- Classical DBMSs are good for formatted data (scalars), short strings, and fixed-length records. Digitized designs are long strings; they have variable-length records or textual information; they often contain vectors and matrices of values.
- Temporal and spatial (see Section 25.3.4) relationships are important in designs for layout, placement, and assembly operations.
- Design data is characterized by a large number of types, each with a small number of instances. Conventional databases have just the opposite situation.
- Schemas *evolve constantly* in design databases because designs go through a long period of evolution.
- Transactions in design databases are of long duration; a designer may “check out” a design object and work on it for several weeks before restoring it (in its modified form) to the database. Similarly, updates are far-reaching because of topological relationships, functional relationships, tolerances, and so on. One change is likely to affect a large number of design objects.

- It is necessary to keep old versions and to create new versions of the same object. A design log must be maintained for tracing the evolution of a design and possibly for backtracking through it.
- Making a design permanent, releasing it to production, archiving it, and so on are specialized functions in a design database.
- Design data must not be duplicated at lower levels of design; since design elements (such as gates in electrical circuits or nuts and bolts in mechanical design) are used in a highly repetitive way, redundancy control is required to suppress certain common attributes. For example, when identical bolts are used at two places, the positional coordinates are stored but other attributes are not repeated. In such cases, a library of design objects can be maintained.

Because of these demands, new data modeling approaches are being proposed to manage VLSI designs, as well as mechanical designs. Object-oriented models are favored because they possess the following characteristics (for a detailed discussion, see Spooner 1986 and Ketabchi 1986):

- *Common model*: The designer's miniworld can be mapped into the database objects by a one-to-one mapping.
- *Uniform interface*: All objects are treated uniformly by accessing them through methods (or user-defined operations).
- *Support of complex objects*: Object-oriented models for engineering designs must allow creation of arbitrarily complex objects involving hierarchies and lattices (Batory and Buchmann 1984).
- *Information hiding and support of abstractions*: The abstraction mechanism can provide the essential external features of objects for design while hiding internal representation or implementation details. Generalization and aggregation are easily supported (see Ahmed and Navathe 1991).
- *Versioning*: A design object may undergo evolution through a series of versions; it may also have different representations. It is necessary to treat a *generic object* and its versions independently. At the same time, there should not be an overuse of the "create version" operation as opposed to an update. ORION (Chou and Kim 1986) supports transient versions and working versions. Ahmed and Navathe (1991) defines a versioning scheme based on intrinsic and extrinsic properties of design objects. A version is created when intrinsic properties are unchanged, but extrinsic properties change. The proposed versioning scheme prevents unnecessary version proliferation.
- *Modularity, flexibility, extensibility, and tailorability*: Object-oriented databases support schema evolution more easily than do conventional models; new objects or new operations can easily be added, and old ones can be modified or deleted.

Early efforts in this direction included a project at the University of Southern California (Afsarmanesh et al. 1985), the ORION system (Kim et al. 1987) at the Microelectronics and Computer Technology Corporation, and an effort at the University of California, Berkeley (Katz 1985). More recently, the ROSE (Relational Object System for

Engineering) system was developed at Rensselaer Polytechnic by Hardwick and Spooner (1989). It is a structurally object-oriented DBMS supporting aggregation, generalization, and association. It has an open architecture with facilities for providing front-end and back-end tools.

Extensions of the relational and network models to accommodate CAD data have been reported but have not been very successful (Wiederhold et al. 1982; Stonebraker et al. 1983). Long transactions and concurrency control of design transactions remain research issues.

Another important development is the standardization of the product data for exchange purposes across vendors, designers, and manufacturers. The PDES (Product Data Exchange using STEP, where STEP is an ISO STandard for the Exchange of Product Model Data) standard is now drawing a very large following (PDES Inc. 1991). It is applied in diverse industries such as building and construction, electrical components, and architecture. An accompanying language called EXPRESS is also gaining popularity as a descriptive language for specifying parts, identifying various types of constraints on them, and providing a uniform way of representing design objects for further analysis/synthesis. The PDES/STEP and EXPRESS standards promote sharing and integration of data for product design. Both of these developments are likely to lead to a new generation of systems and tools in engineering design and manufacturing.

There is also a need for further work on modeling of the design data due to its peculiarities. One promising direction involves the separation of the structure and function in complex physical systems so that each can be independently represented and analyzed (Navathe and Cornelio 1990). This approach further facilitates modeling of the dynamics of such systems using the active database approach (see Section 25.3.1) and aids in simulation (Cornelio and Navathe 1993).

25.2.2 Office Systems and Decision Support Systems

Automation of office work has been one of the fastest-growing areas of application of information systems (Ellis and Nutt 1980). The growing interest and activity in **office information systems** (OISs) is evident from the fact that a journal called *ACM Transactions on Office Information Systems* was launched in 1983 by the Association for Computing Machinery. Subsequently, noting that office information systems are only one specific type of information system, the journal changed its title to *Transactions on Information Systems* in 1990. Database technology is having a major impact on office work because much of it falls in the category of *programmable work*, where the events are predictable and the responses are known. Computers and, in particular, database systems can greatly influence this type of work. The other extreme is *emergent creative work*, according to the taxonomy of Ciborra et al. (1984), where the events are unpredictable and the responses are unknown. The office work of top management, international money traders, and the like falls into this category and requires the assistance of decision support systems. We first briefly point out the differences between information systems and OISs; then we briefly review a data-based model of an OIS; and finally we make a few comments on decision support systems.

Office Information Systems Versus Conventional Database Applications.* Like CAD applications, OIS applications place a higher demand on a DBMS while centralizing the information in a given office environment. This is due to the following characteristics of OIS applications:

- *Semantic richness*: Office data tends to be semantically rich and requires support for unstructured messages, letters, text, annotations, chains of forwarding addresses, oral communications, and so on. There are *stereotypical information groupings* in an office, such as business letters or quarterly progress reports with standardized formats.
- *Time factor*: The time factor and timing constraint must be modeled to capture the following types of information: the total travel time of a document along a path; duration of activities, calendars, and schedules; allowable response time to reply to a message; automatic generation of reminders; and so on.
- *Lack of structure*: Office activities tend to be much less structured than other information systems. Instructions to perform work are incomplete and irregular; constant communication and dialogue are necessary.
- *High interconnectivity*: Compared to a manufacturing workcell that uses machine tools, an office typically represents a much more complex group of elements, each performing a variety of functions and interconnected in multiple directions to elements of different types. It is therefore difficult to model the workflow in an office accurately, and automation and productivity improvements are difficult to achieve via a well-defined office design methodology.
- *Office constraints and evolution*: Because an office is a group of humans, it is subject to constant evolution in which the authorities and job responsibilities of individuals change. It is therefore difficult to model all constraints definitively.
- *Interactive interface*: OISs are highly interactive. It should be possible for even the lowest-level worker in an office to communicate to an OIS. User groups are diverse and span the whole spectrum from naive to sophisticated. This makes interface design challenging.
- *Filtering of information*: Most offices have a built-in hierarchy. It is necessary to filter and summarize information in a pyramid style within the system. Low-level transactions must be summarized before being passed up to the next level of management. Aggregation of data elements should be performed automatically at different prespecified and user-controlled intervals.
- *Priorities, scheduling, reminders*: All of these are important characteristics of different elements of work; a variety of interrupts are generated and must be handled in the office constantly.

It is evident that office automation is not an easy task. A database at the heart of an OIS must be capable of meeting all of the preceding requirements. Because of these factors, we see “islands of mechanization” in offices today rather than a totally automated office. Many conceptual modeling and design methodologies for OISs have been pro-

*This section is largely derived from Bracchi and Pernici (1984, 1987).

posed; examples are OFFIS (Konsynski et al. 1982), OAM (Sirbu et al. 1981), and MOBILE-Burotique (Dumas et al. 1982). Recent research in office information systems has concentrated on modeling the office workflow, defining the database requirements, and dealing with query processing specific to the office setting. An example is the TODOS (Pernici 1989) project, which stands for TOols for Designing Office Systems. Multimedia database technology (see Section 25.3.2) is expected to play a major role in office systems (Masunaga 1987, Woelk 1987, Bertino 1988).

Data-based Model of an Office Information System. Data-based models group data into forms that resemble paper forms in an office. The Office-By-Example project at IBM Research (Zloof 1982) developed a language called Office By Example (OBE) that is an extension of the QBE language described in Chapter 9. OBE models a variety of objects using the basic relational table paradigm: forms, reports, documents, address lists, message lists, menus, and so on. The system allows data types such as image, text, and time. The functions specified on these objects include word processing, querying, and automatic triggering of activities (for example, send letter when payment is more than n days overdue, where n is a run-time parameter or is obtained from some table). Authorization, message communication and forwarding, and document manipulation are examples of other functions handled through the same interface. The main point of data-based office models is to represent the office from the viewpoint of objects manipulated by office workers. The object-oriented approach again seems promising here. Products specific to document processing, electronic multimedia mail, audio/video communication, and the like are appearing for PC Windows-type user environments. One example is the NEWWAVE system of Hewlett-Packard.

Decision Support Systems (DSSs). Office systems or business data-processing systems in general offer decision support to upper-level office workers in several ways:

- *Analysis of office data:* Data must be presented to the user at the right level of detail, using the right display after performing the proper filtering.
- *Controlling the state of the system:* An OIS must have features to determine its own state and to evaluate it. Any discrepancies or exceptional conditions requiring review should be brought to the attention of the right individuals.
- *Support of analytical decision-making tools:* Besides making ad hoc decisions, managers are called on to make long-term strategic and business planning decisions. Alternative choices must be considered and analyzed. A DSS must have knowledge of alternatives as well as the models needed to perform optimization or to reach “intelligent” choices.
- *Support for organizational design and office system design changes:* The DSS should have some features that allow organizational restructuring or information flow modification without disrupting normal operations.

DSSs are being designed today with a DBMS as their central component. In general, a DSS allows the user to perform control and monitoring functions; it must also allow the user to enforce policy decisions (constraints) and actions (procedures). To permit partial automation of decision making in offices, one may resort to an “active” database

system (see Section 25.3.1), where actions can involve generating messages, notices, letters, and so on, or can be more involved in terms of enforcing policies that span the entire set of functions and databases in the office. Various systems have been proposed, but the literature is too vast to be summarized here.

25.2.3 The Human Genome Initiative

Introduction and Goals of the Project. The Human Genome Initiative (HGI) is a good example of a trend in biology that is giving it a new aspect: that of a computational analytical science. The Human Genome Initiative is an international effort aimed at determining the location of the estimated 100,000 human genes that compose the entire human genome. HGI is a monumental scientific adventure that can be compared with efforts in the fourteenth century to draw the most complete and detailed possible map of the earth, given the tools of those days when the earth was believed to be flat. The scientific, medical, and economic implications of the HGI are profound. HGI would ultimately pave the way for a routine decoding of genes that make life possible as well as providing an improved understanding of the diseases afflicting mankind.

The most ambitious goal of the genome project is to complete the most detailed map: a reference sequence of the entire human genome. HGI is a long-term project, with an expected duration of between 25 and 40 years. It involves three key computational challenges: sequence analysis, information storage and retrieval, and protein structure prediction. These elements break down as follows:

- *Sequence analysis problem:* Given a set of DNA sequences, efficient alignment-matching algorithms are needed that can deal elegantly with insertion, deletion, substitution, and even gaps in the series of sequence elements.
- *Storage and retrieval problem:* Effectively storing and retrieving data, and making the information easily accessible to distributed users while effectively dealing with errors, conflicts, and updates is a challenging problem—the more so because the rate at which new sequence information will be generated by the turn of the century will be about 1.6 billion base pairs per year.
- *DNA and protein structure prediction problem:* Within cells, most DNA is associated with proteins that, among other things, influence the torsional stress on the DNA double helix. Under negative torsional stress, the DNA helix may assume alternative structures. Specific sequences favor such structures, and predicting the probabilities of alternative structures for a given sequence is a challenging problem. Another difficulty is the protein-folding problem. After a protein is synthesized by the cell, the protein chain folds according to the laws of physics into a specialized form, based on the particular properties and order of the amino acids. The protein-folding problem is expected to challenge our most powerful supercomputers for the foreseeable future.

In multicellular organisms, DNA is generally found as two linear strands wrapped around each other in the form of a double helix. A DNA strand is a polymeric chain made of nucleotides, each consisting of a nitrogenous base, a deoxyribose sugar, and a phosphate molecule. The arrangement of nucleotides along the DNA backbone is called

the DNA sequence. Four nucleotides are used in the DNA sequences: adenosine (A), guanosine (G), cytidine (C), and thymidine (T). In nature, base pairs form only between A's and T's and between G's and C's. The size of the genome is generally given as its total number of base pairs. A **gene** is a region of a chromosome whose DNA sequence can be transcribed to produce a biologically active RNA molecule. With this background, the goals of HGI related to computing are as follows:

- To establish, maintain, and enhance databases containing information about DNA sequences, location of DNA markers and genes, function of identified genes, and other related information.
- To create maps of human chromosomes consisting of DNA markers that would permit scientists to locate genes quickly.
- To create repositories of research materials, including ordered sets of DNA fragments that fully represent DNA in the human chromosome.
- To determine the DNA sequence of a large fraction of the human genome and that of other organisms.

Requirements of a Genome Database. It is crucial for the scientific community to be able to access information on a topic from a variety of databases that may handle different aspects of the problem. Databases must use standardized or easily translatable formats and must be interconnected. The genome database must be heterogeneous and interoperable. The main problems in achieving these objectives are as follows:

- Different (and seemingly conflicting) views of a sequence must be represented adequately.
- A nucleotide sequence as represented in a database is an imperfect and incomplete picture of an idealized molecule: imperfect, because experimental data always contains errors; incomplete, because there is always a scope to learn more about it; and idealized, because it ordinarily ignores variation.
- The concepts are fluid, and their meanings change all the time. Currently, no good database technology supports a fluid environment of this type.
- The “construction process” of the model of the universe produces inconsistency, ambiguity, and outright contradiction. It is important that the genome database support dynamic schema evolution, while the construction process continues on the basis of the new experimental data and new inferences.

Since the concepts and the schema are continually changing, the query language should be powerful enough to satisfy the needs of the users, each of whom has a different degree of expertise and knowledge. The genome database must perform pattern matching to identify relevant sequences, gaps, and so on, requiring a powerful pattern-matching facility.

The choice and the design of the data model that reflects the biology of the problem are themselves major issues. The relational data model is poorly suited for a genome database because it fails to capture the order in a sequence. Existing object-oriented models are useful in that they can capture complex objects and can perform incremental

consistency checking. However, they have a weak query capability. As we noted in Section 24.7, the LDL system, with its deductive capability, is currently being applied to this problem for the *E. coli* genome.

Current Efforts. Existing databases and repositories that gather, maintain, analyze, and distribute data and materials are already struggling to keep up with the exponential growth of molecular biology. Some databases specialize in map and sequence information from one specific genome; for example, there are databases exclusively devoted to mouse, to *E. coli* bacteria, to drosophila, and to nematode genomes. Others carry particular kinds of information from all the relevant genomes. Some existing U.S. databases and repositories are given next. More information on these databases is summarized in Table 25.2.

On-line Mendelian Inheritance in Man (OMIM). This is a computerized atlas of human traits that are known to be inherited—that is, the expressed genes.

Human Gene Mapping Library (HGML). This library consists of five linked databases—one each for map information, relevant literature, RFLP maps, DNA probes, and contacts (researchers with information on data and materials).

GenBank. This is the major U.S. database for nucleic acid sequence information from humans and other organisms.

Protein Identification Resource (PIR). This resource contains sequence data for proteins and amino acids, with annotations that indicate known functional regions.

Protein Data Bank (PDB). This library gathers information on the atomic structure of nucleic acids, messenger RNA, amino acids, proteins, and carbohydrates that has been derived from crystallographic studies.

25.3 Next Generation of Databases and Database Management Systems

In this section we will briefly survey the major types of databases and the associated database technology that is currently under development. This includes the following: active databases (Section 25.3.1), multimedia databases (Section 25.3.2), statistical and scientific databases (Section 25.3.3), and spatial and temporal databases (Section 25.3.4). This is followed by a section on unified and extensible database management (Section 25.3.5), where we survey two extensible systems—EXODUS and GENESIS—and a commercial product called UniSQL, which unifies the object-oriented and relational approaches to database management.

25.3.1 Active Databases*

Traditionally, DBMSs are passive; they execute queries or transactions only when explicitly requested to do so by a user or an application program. But many applications, such as process control, power generation/distribution networks, automated office workflow

*This section is largely based on Chakravarthy 1990a.

Table 25.2 Some Existing Genome Databases

Name	Institution	Contents	Format	On-line Access
GenBank	Los Alamos National Laboratory	DNA	MT/CD-ROM	on-line/anonymous FTP
EMBL	European Molecular Biology Laboratory	DNA	MT/CD-ROM	N/A
PIR	National Biomedical Research Foundation	protein	MT/CD-ROM	on-line
SWISS-PROT	Geneva University	protein	MT/CD-ROM	anonymous FTP
PDB	Brookhaven National Laboratory	protein	MT	anonymous FTP
GDB/OMIM	Johns-Hopkins University	generic map	CD-ROM	on-line
MEDLINE	National Library of Medicine	bibliography	CD-ROM	on-line

control, program trading, battle management, and hospital patient monitoring, which require timely response to critical situations, are not well served by these “passive” DBMSs. For these **time-constrained** applications, conditions defined on states of the database must be monitored, and once these conditions occur, specified actions must be invoked, possibly subject to some timing constraints. A possible scenario in automated manufacturing involves monitoring for an event—widgets coming out of an assembly line machine M1 over an interval—evaluation of a condition—10% of widgets being defective over the time interval—and taking one or more actions—bringing the machine M2 on-line and notifying appropriate personnel. All of this may involve access to shared databases that are constantly being updated by several users and have to be maintained in a consistent state.

Approaches. Traditionally, two approaches to meeting the requirements of time-constrained applications have been taken (see Figure 25.2). The first is to write a special application program that polls (periodically queries) the database to determine whether the situation being monitored has occurred. The second is to augment each program that updates the database to check the situation being monitored. The first is difficult to implement because the optimal polling frequency is hard to determine; the second compromises modularity and reuse of code. Active databases support condition monitoring (including triggers and alerters) at a level of abstraction that has three characteristics: its semantics are well-defined; it satisfies the modeling and efficiency requirements of nontraditional database applications; and it seamlessly integrates with a DBMS.

An active DBMS continually monitors the database state (including the system clock) and reacts spontaneously when predefined events occur. Functionally, an active database management system monitors *conditions* triggered by *events* representing data-

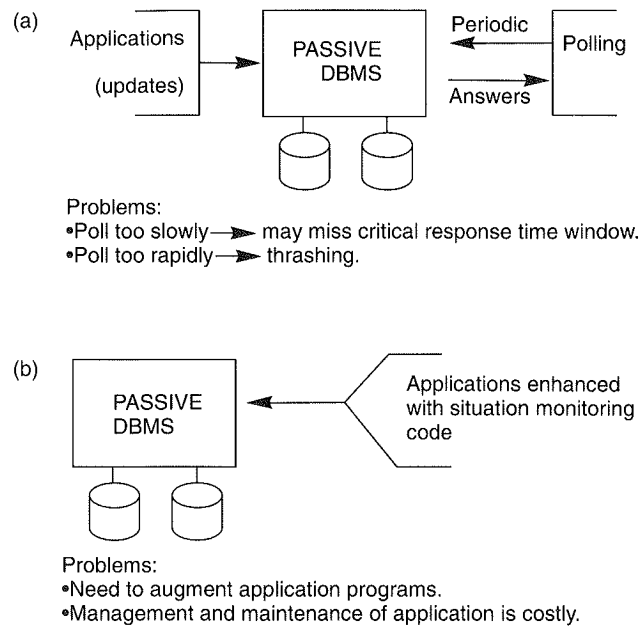


Figure 25.2 Conventional passive DBMS mimicking active behavior: (a) the polling approach; (b) the enhanced application approach.

base events (for instance, updates) or nondatabase events (for instance, hardware failure detected by a diagnostic program); and if the condition evaluates to true, the *action* is executed. An active DBMS provides both modularity and timely response. The integrated approach to an active DBMS illustrated in Figure 25.3 can be viewed as the next logical step beyond deductive database systems. In an active DBMS, the notion of an event is generalized and made explicit. Rules include events, conditions, and actions. Figure 25.4 shows an active database for automatic reordering of items in an inventory control application.

Issues in Active Databases. Six important issues set active DBMSs apart from passive DBMSs:

- **Efficiency:** The set of all event–condition–action rules is likely to form a large set of predefined queries that need to be efficiently managed and evaluated when specified events occur. Evaluation of complex rules under time constraints is a challenge when sets of rules that represent a highly dynamic environment become very large. This opens new possibilities for optimization by using results from multiple-query optimization (Chakravarthy 1991; Sellis 1993).
- **Modes of rule execution:** Rules can be fired and executed under different modes: immediate, deferred, or detached. Under immediate mode of execution, the processing of the remaining steps of the original transaction (that is, the triggering

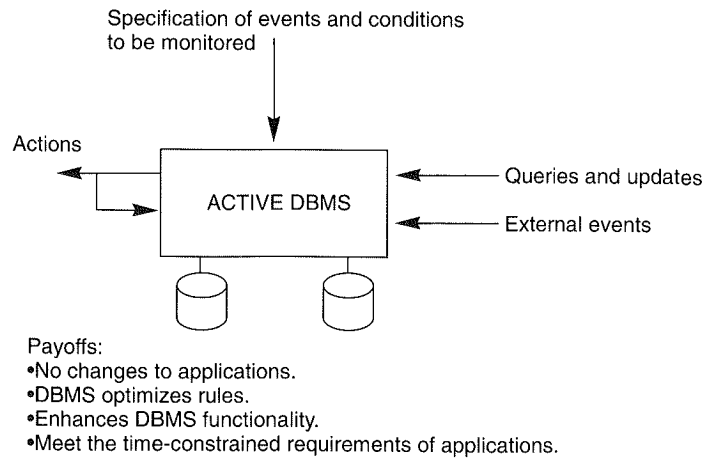
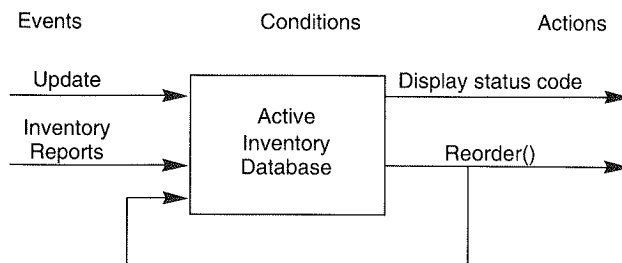


Figure 25.3 The integrated approach to active databases.

transaction that caused the event to occur) is suspended until the fired rule has been completely processed. Response time and concurrency can be improved if the condition evaluation or action execution is detached from the original transaction (that is, run in a separate transaction). If a rule is allowed to trigger another rule as a part of its execution, the transaction model becomes nested, leading to further complexity.

- **Data model extensions:** Active DBMSs require enhancements to the data model in at least two ways: for specifying events, and for specifying conditions and actions. In addition to events that correspond to database operations (such as insert, delete, and modify), temporal and/or periodic events (for example, that the balance in all bank accounts should be checked at 5 P.M. every day), and user- or



EVENT: Update Quantity on Hand (item).

CONDITION: $\text{Quantity_on_Hand}(\text{item}) + \text{Quantity_on_Order}(\text{item}) < \text{Threshold}(\text{item})$.

ACTION: Reorder (item).

Figure 25.4 Active DBMS example (from Chakravathy 1989).

application-generated events (such as a failure signal from a diagnostic routine on a hardware component) need to be supported. The language can be made to support a specification of complex events based on an event algebra (Jagdish and Gehani 1992).

- **Management of rules:** First, the ability to manipulate rules (add/delete/modify) as if they were any other data object in the system is essential. Second, mechanisms for enabling and disabling individual rules or rule sets are often needed. For example, the set of rules activated while an aircraft is taxiing needs to be disabled when it becomes airborne, and a different set of rules for the current context needs to be activated. Finally, selective enabling and selective disabling of rules are also important.
- **Supporting DBMS functions:** DBMS functions can be supported by the active DBMS. Constraint management (including integrity and security enforcement), maintenance of derived data (such as views), and rule-based inferencing are a few examples.
- **Interaction with parts of DBMS:** Unlike in a conventional query optimizer, in an active DBMS, rules cannot be optimized in isolation. Optimization of rules requires interaction with several components of a DBMS (such as the transaction manager, the object manager, and the scheduler).

State of the Art of Active Databases. Following is a brief overview of the research and commercial efforts in the active database area.

HiPAC (Chakravarthy 1989, 1990), which stands for High Performance ACtive database system, was a research effort focussing on active time-constrained data management. The knowledge model of HiPAC included primitives for events, conditions, and actions, and it formulated the notion of “rules as first-class objects.” In the execution model, the interaction between rule execution and conventional transaction execution was investigated; coupling modes and algorithms for supporting rules that had various coupling modes were developed. Various other issues were addressed as part of the HiPAC project: an architecture of an active DBMS; combining real-time processing with databases; multiple query optimization; and modeling and performance evaluation of various coupling modes. For the battle management application, PROBE (Dayal 1985) was extended to include event detection and processing of conditions consisting of spatial queries.

An Event/Trigger Mechanism (ETM) was designed at the University of Karlsruhe (Dittrich 1986; Kotz 1988) to enforce complex consistency constraints in design databases. The effort addressed the general problem of constraint enforcement and was implemented on top of the DAMASCUS design database system. Event, action, and trigger tables are maintained for efficient access and determination of triggers when a specific event occurs. The active capability is an add-on to the underlying system, and consequently the modes in which a trigger can be executed with respect to a transaction are limited. In POSTGRES (Stonebraker 1987), commercially designated Miro, database operations and other predefined events (such as date) trigger the rule evaluation. Rules are expressed in QUEL by using keywords (such as *always* and *demand*), and the system uses

special types of locks for the database items mentioned in the condition for the implementation of rules. Trigger implementation (also called *tuple-level processing*) is embedded in the executor and uses three types of locks. Query rewrite uses the well-known query modification approach outlined in Chapter 17. Only the equivalent of immediate coupling mode is supported.

The Starburst project at IBM has addressed the problem of set-oriented rules (Widom 1990) triggered by database state transitions in the context of relational databases. Events corresponding to database operations (inserted/deleted/updated) can be associated with a condition and an action by using the SQL syntax. The interaction between transaction execution and rule execution is addressed in detail.

Sybase (1992) supports simple triggers, where the triggering conditions involve only a single relation, and can be viewed as being an active DBMS in principle. Sybase recognizes events that correspond only to database operations—namely, insert, delete, and update. Triggers cannot return data to the user and cannot be specified on views or temporary objects. One trigger cannot fire another trigger, and triggers are always executed in immediate mode.

InterBase (Interbase 90), on the other hand, does not impose most of the restrictions seen in Sybase. In InterBase, a trigger specification consists of an event, a sequence number associated with the trigger, and a trigger action. An event can be one of the database operations (store, erase, or modify). Any number of triggers can be defined on a relation. The user can also specify whether the trigger action is to be executed before or after the operation corresponding to the event is performed. Multiple rule execution is supported by using the priority information (an integer) to resolve conflicts. Access to new and old tuple values is provided through the keywords *old* and *new*.

ORACLE version 7, INGRES, INFORMIX, and so on all provide some degree of rule and trigger support. In terms of commercial DBMS products, rule-based facilities are likely to be offered with increasing frequency in the coming years.

25.3.2 Multimedia Databases

Multimedia databases store information that originates from different media, including numeric data, text, bitmap images, vector graphics images, audio, and video. They also use appropriate display devices in presenting the information. Large-scale applications of multimedia databases can be expected in the following disciplines in the coming years:

- *Documents and records management*: A large number of industries and businesses keep very detailed records and a variety of documents. The data may include engineering design and manufacturing data, medical records of patients, publishing material, and insurance claim records, depending on the organization.
- *Knowledge dissemination*: The multimedia mode is a very effective mode of knowledge dissemination. As such, we hope to see electronic books and repositories of information on many topics for public consumption.
- *Education and training*: Teaching materials for different audiences, from kindergarten children up to professional equipment operators, can be designed from multimedia sources.

- *Marketing, advertising, retailing, and travel:* There are virtually no limits to using multimedia information in these applications for effective presentations.
- *Real-time control and monitoring:* Coupled with active database technology (see Section 25.3.1), multimedia presentation of information can be a very effective means for monitoring and controlling manufacturing operations, nuclear power plants, patients in intensive care units, transportation systems, and so on.

Issues in Multimedia Databases. Multimedia applications dealing with thousands of images, documents, audio and video segments, and free text data depend critically on appropriate modeling of the structure and content of data. Multimedia information systems are very complex and embrace a large set of issues, including the following:

- *Modeling:* This area gets into the controversy of applying database versus information retrieval techniques to the problem. There are problems of dealing with complex objects (see Chapter 22) made up of types of data that range over numeric, text, graphic (computer-generated image), animated graphic image, audio stream, and video sequence. Documents constitute a specialized area and deserve special consideration.
- *Design:* The conceptual, logical, and physical design of multimedia databases has not been researched yet. It can be based on the general methodology described in Chapter 14, but the issues at each level are far more complex.
- *Storage:* Storage of multimedia data on standard disklike devices presents problems of representation, compression, mapping to device hierarchies, archiving, and buffering during the I/O operation. Adhering to standards such as JPEG or MPEG is one way most vendors of multimedia products are likely to deal with this issue. In DBMSs, a “BLOB” (Binary Large Object) facility allows untyped bitmaps to be stored and retrieved. Standardized software will be required to deal with synchronization, compression/decompression, and so on, which are still in the research domain (Vin 1992).
- *Retrieval:* The “database way” of retrieving information is based on query languages and internally kept index structures. The “information retrieval” way of dealing with it relies strictly on keywords or predefined index terms. For images, video data, and audio data, this opens up many issues. We discuss a few of them later.
- *Performance:* For multimedia applications involving simply documents and text, performance constraints are subjectively determined by the user population. For applications involving video playback or audio-video synchronization, however, physical limitations dominate. For instance, video must be delivered at a steady rate of 60 frames per second. Techniques for query optimization may compute expected response time before evaluating the query. The use of parallel processing of data may alleviate some problems, but currently such efforts are largely experimental.

The preceding issues have given rise to a variety of open research problems. We only address a few of them as representative problems in this section.

Databases Versus Information Retrieval Perspectives. In database models and systems, modeling the data content has not been an issue because the data has a rigid structure and the meaning of a data instance can be inferred from the schema. On the other hand, information retrieval (IR) is mainly concerned with modeling the content of text documents (by keywords, phrasal indexes, semantic networks, and so on) for which structure is generally neglected. By modeling the content, the system can determine whether a document is relevant to an inquirer's query by examining the content-descriptors of the document. Consider, for instance, an insurance company's accident claim report as a multimedia object: it contains images of the accident, structured insurance forms, audio recordings of the parties involved in the accident, the text report of the insurance company's representative, and other information. What data models to employ in representing multimedia information remains an open issue.

Requirements of Multimedia/Hypermedia Data Modeling and Retrieval. To capture the full expressive power of multimedia, the system should have a modeling construct that lets the user specify links between any two arbitrary nodes. **Hypermedia links** can take on a number of different forms, including the following:

- Links can be specified with or without associated information. A link may have large descriptions associated with it.
- A link can start from a specific point inside a node, or it can start from the whole node.
- A link can be directional or nondirectional.

The link capability of the data model should account for all of these variations. When a content-based retrieval of multimedia data is needed, the query mechanism should have access to the links and the link-associated information. The system should provide facilities for defining views over links, as well as private and public links. Valuable contextual information can be obtained from the structural information. An automatically generated hypermedia link does not reveal anything new about the two nodes, because the link was created automatically by some method. On the other hand, manually generated hypermedia links and the link information can be utilized to gain more knowledge about the nodes that are being connected. Facilities for creating and utilizing such links, navigational query languages to utilize the links, and so on are important features of any system permitting effective use of multimedia information.

Indexing of Images. There are two approaches to the problem of indexing images. One uses image-processing techniques to identify the different objects automatically. The other uses manual indexing techniques to assign index terms and phrases. One important problem in using image-processing techniques to index pictures relates to scalability. The current state of the art allows only simple patterns in images. The complexity increases with the number of recognizable features. Another important problem relates to the complexity of the query. Rules and inference mechanisms, as discussed in Chapter 24, can be used to derive higher-level facts from simple features of images. This allows high-level queries like "find the apartment designs that allow maximum sunshine in the living room," in an architectural application.

The information retrieval approach to image indexing is based on one of the following three indexing schemes:

- *Classificatory systems* (Gordon 1988) classify the images hierarchically into certain predetermined categories. In this approach, the indexer and the user should have a good knowledge of the available categories. Finer details of a complex image and relationships among objects in an image cannot be captured.
- *Keyword-based systems* use an uncontrolled (alternately, a controlled) vocabulary as in indexing of textual documents. Simple facts represented in the image (like “ice-capped region”) and facts derived as a result of high-level interpretation by humans (like permanent ice, recent snowfall, and polar ice) can be captured.
- In the *entity-attribute-relation scheme* (Leung 1992), all the objects in the picture and the relationships between objects and the attributes of the objects are identified.

In the case of text documents, a human indexer can choose the keywords from the pool of words available in the document to be indexed. This is not possible in the case of image and video data.

Open Problems in Text Retrieval. Although text retrieval has existed in business applications and library systems for a long time, the following open problems still confront the designers of text retrieval systems:

- *Phrase indexing:* Substantial improvements can be realized if phrase descriptors (as opposed to single-word index terms) are assigned to documents and used in queries, provided that these descriptors are good indicators of document content and information need (Fagan 1987).
- *Use of thesaurus:* One reason for the poor recall of current systems is that the vocabulary of the user differs from the vocabulary used to index the documents. One solution is to use a thesaurus to expand the user’s query with related terms. The problem then becomes one of finding a thesaurus for the domain of interest.
- *Resolving ambiguity:* One of the reasons for low precision (the ratio of the number of relevant items retrieved to the total number of retrieved items) in text information retrieval systems is that words have multiple meanings. One way to resolve ambiguity is to use an on-line dictionary (Krovetz 1992) while the other is to compare the contexts in which the two words occur (Salton 1991).

Multimedia information systems promise to bring about a marriage of the disciplines of information retrieval and database management, which have heretofore tended to diverge.

25.3.3 Scientific and Statistical Databases

The use of computers to manage scientific and statistical information is not new. However, database technology was rather late in reaching statisticians and scientists. In March, 1990, the National Science Foundation sponsored a workshop, hosted by the University of Virginia, at which representatives from the earth, life, and space sciences

gathered together with computer scientists to discuss the problems facing the scientific community in the area of database management. These problems are likely to grow exponentially over the next decade. We summarize in this section the important issues raised in this report (French et al. 1990).

Scientific data is unique because of its relatively static nature and indefinite retention. The typical transaction-processing environment of conventional databases is absent for scientific databases; they have a low update frequency, and old data is rarely discarded. We referred earlier to the large volume of scientific data in the human genome initiative: it involves 3 billion nucleotide bases. The Magellan planetary probe will generate about 1 trillion bytes of data over five years.

Scientific database systems comprise three “dimensions”: level of interpretation, intended analysis, and sources of information. In the following discussion, *data set* is used to mean data related to a single experiment or mission; *database* is used for any aggregate of data that is summarized or compiled from multiple experiments.

The Interpretation Dimension. This dimension refers to the continuum reflecting the raw (versus processed) nature of the information. It affects what one expects of the data set and how the data set is used. Based on this dimension, data can be divided into the following categories:

- *Raw/sensor data:* This consists of (seldom saved) raw values obtained directly from the measurement device.
- *Calibrated data:* This consists of (normally preserved) raw physical values, corrected with calibration operators.
- *Validated data:* This consists of calibrated data that has been filtered through quality-assurance procedures; it is the most commonly used type of data for scientific purposes.
- *Derived data:* This consists of frequently aggregated data, such as gridded or averaged data, for which the detail of the underlying measurements has been lost.
- *Interpreted data:* This consists of derived data that is related to other data sets or to the literature of the field.

Typically, information about the processing must be retained and propagated together with a data set. The raw data and its summary may need to be interrelated.

Intended Scientific Analysis. Most stored scientific data is subjected to some type of analysis. Earth science data is subjected to statistical, time-series analysis. Genome data is analyzed for patterns over character storage. Space data in large arrays is transformed (for example, by using the Fourier transform) for spectral analysis. The relational data model is unsuitable because of its inability to represent sequences and because of the lack of representation for characteristics of data used during analysis. Sometimes, data is to be “corrected” rather than “updated,” but there are no facilities for accomplishing this in relational DBMSs.

Sources of Data. This is a fundamental discriminatory feature of scientific databases. In a single-source database environment, raw data is collected and further processes by fil-

tering or calibration techniques; the resulting validated data is used in light of the scientific mission. The syntactic and semantic complexity of the interpreted data is much higher than that of the original data. Multisource databases add a new dimension of complexity, with multiple methodologies of data collection, instruments for data recording, and different protocols affecting encoding and accuracy. These databases may be independently used and managed by diverse computing systems and DBMSs. However, the goal typically is to create a common data archive such as GENBANK for the human genome initiative (see Section 25.2.3), to be used by the original research teams that are generating and managing the raw data. This inherits the problems of distributed and multi-databases we discussed in Chapter 23.

Traditional DBMSs have proved inadequate to deal with the above set of requirements mainly for the following reasons:

- They lack features to deal with the built-in redundancies and inaccuracies of data during scientific experimentation and its recording. The progression of data along the interpretation dimension cannot be handled gracefully.
- Analysis of data using multivariate statistical functions and aggregations is lacking.
- DBMSs have the overhead of transaction processing, which is unnecessary for most scientific applications.
- The inherent distinction in statistical data between a value and what it stands for (such as a number (value) representing the number of households in Fulton County that have an income in the range of \$75,000–\$100,000 per year and at least one child in a private school) is not supported in traditional DBMSs. Abstractions such as the cross-product proposed in the data model OSAM* (Su et al. (1988)) are useful.

Important issues in the areas of data definition, data manipulation, and technology standards need to be addressed to meet the requirements of scientific and statistical databases. In the area of data definition, the following issues arise:

- New data models, including those covered in Chapter 21, need to be explored. They should support special data types such as time series and DNA sequences.
- Meta-data management is crucial, with a provision needed for dealing with citations, different types of documentation, and experimental details.
- Evolution of data must be supported with an appropriate audit trail.
- Standards are needed across scientific disciplines for recording experimental results and even for sharing citations.

In the area of data manipulation, the following points are relevant:

- It is not adequate to perform conventional querying and report generation. Statistical and analytical functions must become an integrated part of the system. A trend is also visible to add a relational querying language interface such as SQL to existing statistical packages such as SAS.
- Various summarization and aggregation operations, together with a facility to describe the summary statistic, must be provided.

- Export and import features must exist for transferring data into and out of the scientific database and into statistical utilities.
- Since a typical statistician or scientist is a novice database user initially, a range of user interfaces and browsing and navigational facilities must be provided. Provision should be made to allow users to “learn” the data management aspect of the system through a graduation of those features and interfaces.

In the area of technology standards, the following issues are prominent:

- Physical encoding of data is important, with compression and indexing techniques needed for efficient storage and retrieval. Encryption of data is essential for privacy where human subjects are involved.
- For long-term experiments (such as a space probe or long-term tracking of patients with a certain disease), schemas may have to undergo changes as experimental techniques evolve.
- Standards including data formats and associated analysis tools are needed within disciplines and across disciplines. An example is the FITS standard used by the astrophysics community.
- Transmission of entire data sets over high-speed networks is becoming feasible. Extensive catalogs are needed with names, aliases, codes, data derivation procedures, data quality-control measures, and usage information. Appropriate archiving of data sets is necessary.

It is evident from the preceding discussion that the community of scientific database users has a very special set of needs. French et al. (1990) contains a series of recommendations that include addressing sociological as well as technical problems; professional societies are called upon to encourage cooperation among scientists through easier exchange of information.

25.3.4 *Spatial and Temporal Database Management*

In this subsection we cover two important types of data that have started receiving attention from database modeling researchers only recently. In Section 25.2.1, while discussing CAD/CAM applications, we alluded to geometric modeling. Whether in the context of CAD/CAM, of architectural and civil engineering design, or of cartography and geological surveys, it is important to model the spatial dimension. Today's data models are lacking in this area. The spatial semantics can be captured by three common representations:

- *Solid representation*: The space is divided into pieces of various sizes. The spatial characteristics of an entity are then represented by the set of these pieces associated with that entity.
- *Boundary representation (or wire frame models)*: The spatial characteristics are represented by line segments or boundaries.
- *Abstract representation*: Relationships with spatial semantics, such as ABOVE, NEAR, IS-NEXT-TO, and BEHIND, are used to associate entities.

The PROBE project at Computer Corporation of America (Dayal et al. 1987) provided support for spatial data in the functional data model DAPLEX (Shipman 1981). While it is clear that many applications would benefit from such facilities, different application domains have varying requirements. For example, in VLSI applications, space is two-dimensional and discrete; basic objects to be stored are points, line segments, rectangles, and polygons. In solid modeling for most manufacturing applications, space is three-dimensional and continuous, and the basic objects are parametrically defined curves and surfaces. The operations relevant to each application area also differ.

If we assume that a 2D or 3D space is represented by using line segments and polygons, the following representative queries might be made:

- Queries about line segments:
 - All segments that intersect a given point or set of points.
 - All segments that have a given set of endpoints.
 - All segments that intersect a given line segment.
- Proximity queries:
 - The nearest line segment to a given point.
 - All segments within a given distance from a given point (also known as a range or window query).
- Queries involving attributes of line segments:
 - Given a point, the minimum enclosing polygon whose constituent line segments are all of a specified type.
 - Given a point, all the polygons that are incident on it.

Different ways of storing objects in space have been proposed by decomposing space. Examples include the following:

- *Minimum bounding rectangles*: Objects are grouped into hierarchies, which in turn are organized into structures similar to B-trees. Examples include R-tree (Guttman 1984) and R*tree.
- *Disjoint cells*: Objects are decomposed into subobjects. Each subobject is in a different cell.
- *Uniform grid*: The space is divided by overlaying a uniform grid structure. Objects appear in a specific cell of this structure.
- *Quad trees*: This is a hierarchical variable-resolution data structure; there are many variations of it.

The representation of objects in space and the processing of queries on those objects are specific areas of investigation that are being heavily pursued by the developers of Geographic Information Systems (GISs). For a thorough review of spatial data structures and their applications, the reader should consult Samet (1990a, 1990b).

Temporal Data Management. Temporal information is a special one-dimensional case of spatial information. Temporal aspects built into databases must include three types of

support for time: time points, time intervals, and abstract relationships involving time (before, after, during, simultaneously, concurrently, and so on). The **history** aspect of databases is very important for applications such as project management, patient histories in hospitals, maintenance histories of equipment, and administrative and operational control in office systems.

One limitation of current databases is that information becomes effective at the same moment as it is recorded in the database. There are no provisions for making a distinction between the *registration time* (or *transaction time*) when certain data is entered and the *logical time* (or *valid time*) period during which the specific data values are valid. In fact, sometimes a transaction time must be recorded for an event. Many database updates in real applications are either *retroactive* (they became effective at some previous point in time) or *proactive* (they become effective at some time in the future). Another limitation is that current systems maintain no history of changes. Each update destroys old facts. The database thus represents only the current state of some domain rather than a history of that domain. To deal with the preceding requirements, data models are needed that explicitly incorporate time, separating time-varying information from time-invariant information and representing them separately. One proposed model is the Temporal Relational Model (TRM) of Navathe and Ahmed (1989). In this model, attributes are divided into time-varying and non-time-varying ones, and so are relations. In time-varying relations, two timestamp attributes are appended, representing the start time and the end time of the logical time over which the tuple is valid. This approach is called *tuple time stamping*. Figure 25.5 shows a time-varying EMPLOYEE relation in TRM.

For this model, a time normalization procedure is proposed to avoid temporal anomalies; the SQL query language is extended into Temporal SQL (TSQL) with a variety of features, including the WHEN clause, temporal ordering, time-slice, aggregate functions, grouping, and a moving-time window. Another school of thinking proposes *attribute time stamping*, which appends a new attribute value with a timestamp at each update time (Ben-Zvi 1982; Clifford and Tansel 1985; Gadia 1988). This results in unnormalized relations with a much more complicated algebra and query language; furthermore, relationships involving non-time-varying keys *cannot* be represented in this approach.

A recent landmark in temporal database research was the publication of a comprehensive collection of works representing the state of the art in this field, edited by Tansel

EmpNo	Salary	Position	T _s	T _E
33	20K	Typist	12	24
33	25K	Secretary	25	35
45	27K	Jr. Engineer	28	37
45	30K	Sr. Engineer	38	42

Notes: T_s: Time start

T_E: Time end

Figure 25.5 A time-varying relation in the Temporal Relational Model.

et al. (1993). The organization of this collection indicates topics in which interesting work has appeared in temporal database management in the recent past:

- *Extensions to the relational model:* This work has emphasized the different ways of adding timestamp information to a static relational model that contains “snapshot” relations. Different types of time may be added by using the two primary approaches: triple timestamping, which was illustrated earlier; and attribute timestamping, which gives rise to unnormalized (non-1NF) relations. Extensions to SQL and QUEL have been proposed to handle temporal operations. Temporal relational algebras, calculi, and aggregate functions have been proposed. An organized effort is underway to “standardize” temporal enhancements to SQL.
- *Other data models:* This work includes extensions to the ER model and to the functional data model. A spreadsheet history model and a temporal deductive database model have also been proposed.
- *Implementation issues:* Query processing and optimization take a new form, depending on the data model and the language used. Indexing of temporal information in various ways (Elmasri et al. 1990) and updating as well as concurrency control and recovery of *transaction-time* (or *rollback*) databases have been investigated. The temporal join operator may assume many forms; its optimization has been studied. Finally, incremental and decremental computation of transaction-time databases has been investigated.

Open problems include reasoning with temporal information, processing transactions over valid-time and transaction-time databases, mixing temporal processing with active and deductive databases, and integrating temporal information over heterogeneous environments.

With the advent of optical write-once disks that can store 10^{13} bytes of data, appending updates to history data becomes viable. Further work in the implementation aspects and commercial feasibility of these models is necessary, however. Implementation of these data models also places demands on operating systems for time management, recording, and synchronization of multiuser time frames.

25.3.5 Extensible and Unified Database Management

As we saw earlier in this chapter, there is a need to build new DBMSs to meet the challenges of the new applications. DBMSs are complex pieces of software with complicated algorithms and relationships among components. Because it takes a long time to develop a DBMS, a new modular approach has been proposed that builds DBMSs out of “DBMS parts” or building blocks.

Such an **extensible database management system** would assemble prewritten modules (algorithms) into new “tailor-made DBMSs.” This approach has several obvious advantages:

- New DBMS development is rapid and economical.
- Technological or algorithmic improvements can quickly be incorporated into the reusable modules, and as a result, the system can always remain up to date.

- Proposed new techniques and algorithms can be evaluated without making significant modifications to the system.

This approach has been pursued by the GENESIS project at the University of Texas (Batory et al. 1986) and by EXODUS at the University of Wisconsin (Graefe and Dewitt 1987; Carey et al. 1986, 1986a, 1988). GENESIS takes the approach of defining the components of a DBMS and the interfaces among them in such a way that a new DBMS can be configured within minutes from a menu-driven user interface, just by selecting appropriate options. Plug-compatible modules are designed for access methods, query optimization, concurrency control, recovery, and other functions by preimplementing a set of alternative algorithms. The common interfaces allow them to be pieced together as building blocks.

EXODUS is architected differently. It provides certain kernel facilities, including a versatile storage manager and a type manager. The type manager permits definition of class hierarchies with multiple inheritance. The storage object is an untyped, uninterpreted variable-length byte sequence of arbitrary size. Buffer management, concurrency control, and recovery mechanisms are provided on top of the storage manager and can be modified. Type-independent index structures, including B+ trees, grid files, and linear hashing, can be selected from a library of access methods. The language E is provided for the database implementor (DBI). It extends C by adding the notion of persistent objects to C, and it frees the implementor from worrying about the internal structure of those objects. The “operator methods” layer in the architecture contains a mix of DBI-supplied and EXODUS-supplied code to operate on the typed, stored objects. Query processing is divided into query optimization and query evaluation. The DBI supplies the description of operators (in a target query language), methods to implement the operators, and cost formulas for those methods. The rule-based optimizer generator transforms these descriptions into C source code, which constitutes a tailored optimizer for the target system. The EXODUS storage manager has been used by various vendors (including O2, which we discussed in Chapter 22) to implement their own DBMSs.

The preceding approach to configuring, which involves generating a tailored DBMS, contrasts with another approach proposed to deal with new applications: that of building DBMSs with extensible functionality by providing a wide set of features. This **full-functionality approach** is exemplified by the PROBE (Dayal et al. 1987) and STARBURST (Haas et al. 1988) projects.

We have already referred to PROBE and STARBURST, while discussing active databases; and we mentioned spatial query processing in PROBE. The POSTGRES DBMS, now called Miro (Stonebraker et al. 1992) also combines object-oriented and active database capabilities with the relational model.

A third trend in this category is toward unified database management. An example of a system in this category is UniSQL, which claims to combine “the sophistication, power, and ease of use in the popular application development tools” while “delivering on the promise of object-oriented development and multimedia database integration.” The UniSQL product is organized as follows:

- *UniSQL/X*: This component provides the client-server DBMS platform. It allows traditional relational applications to coexist with the new object-oriented database application. The *SQL/S* is an ANSI SQL-compliant object-oriented query language.

- *UniSQL/M*: This component is a heterogeneous database manager that allows access to databases from other relational and prerelational DBMSs. It supports an object-oriented query language, SQL/M, that allows the definition and processing of a multi-database schema.
- *UniSQL/4GE Tools*: Object Master is a tool for dynamically generating applications for the creation and management of objects within SQL/X and SQL/M environments. Other tools, called Visual Editor and Media Master, allow for viewing and editing of schemas and for sophisticated report generation.

The next generation of DBMSs is likely to be patterned after the UniSQL DBMS. They will combine features of relational and other data models and provide a combination of the active, multimedia, scientific, spatial, and temporal facilities we have discussed so far.

25.4 Interfaces with Other Technologies and Future Research

Database technology is not developing in isolation. It is very closely tied to several other disciplines, some of which have already been pointed out. For example, the distributed and multi-database area (Chapter 23) is closely tied to telecommunications and networking. Multimedia databases are closely tied to information retrieval. In this section, we comment on a few such interfaces and some likely areas for future development.

25.4.1 Interface with Software Engineering Technology

The primary goal of software engineering is to develop ways to make the software development process easier, more effective, and more efficient. A prime end product of the software engineering activity is application software. With DBMSs becoming so commonplace, the application software development should be considered in the context of DBMSs. A merging of the concepts from both of these disciplines should occur in the following areas:

- *Design databases*: A large design activity such as developing a large software system is like large design projects in other areas, such as building a skyscraper, a power station, or a manufacturing facility. A large system design can be hierarchically decomposed into designs of smaller and smaller components. Each component evolves, design specifications change, and new design alternatives appear. A proper way to consolidate and track the design information is to create a database for it.
- *Application generation from high-level specification*: This area is of great interest in commercial application development. Several commercial application generators are on the market, as part of CASE (computer-assisted software engineering) tools. The main incentive is that such generators reduce the effort and cost of application development. Since this goal is shared by DBMSs, in the future we are likely to see application generators closely coupled with underlying DBMSs.

- *Software tools*: This is an important area of future development. Software tools have two purposes: to reduce the drudgery of certain tasks for humans, and to improve the performance of certain machines. Various types of tools are needed with DBMSs. Tools of the first type include requirements specification and analysis tools, conceptual design tools, tools for view integration, tools for schema mapping among models, tools for physical design and optimization, and partitioning and allocation tools. Tools of the second type include performance monitoring, tuning, reorganizing, and restructuring tools. The past few years have seen tremendous activity in tool development. However, few current tools provide good interfaces and can interface easily with other tools. Of course, there are a few exceptions. Vigorous activity in tool development is likely to continue in the next few years.
- *Prototyping*: This is a popular topic among software engineering enthusiasts, but it has received little attention in the database area. Prototyping of databases and applications can go a long way toward validating the design of schemas, refining their structures, and evaluating the relative frequencies of queries (transactions). We are likely to see substantial development in this area. Increasing availability of personal computers and workstations makes the role of prototyping even more significant.

25.4.2 Interface with Artificial Intelligence Technology

In Chapter 24, we discussed the relationship between logic and databases and saw how it led to the development of the deductive database technology. Today there is growing interest in moving from databases to **knowledge bases** (KBs). Knowledge is information at a higher level of abstraction. For example, “Mr. Jones is 45 years old” may be considered to be a fact in a database. “Mr. Jones is middle aged” is not such a precise fact; it is a broader, higher form of knowledge. Similarly, while “Mr. Jones had an accident on January 17, 1988, in New York at the ramp leading in from I-495 to I-278” is a fact, “Mr. Jones is a reckless driver” is knowledge. “All middle-aged men are reckless” is another higher-level element of knowledge, although it may represent a poor inference because it is not generally correct.

It is very hard to define and quantify knowledge. Knowledge is typically generated by experts in some domain of expertise. Thus knowledge is used to define, control, and interpret the data in a database.

Knowledge comes in various forms (Wiederhold 1984):

- *Structural knowledge*: This is knowledge about dependencies and constraints among the data (for example, “insertion into benefit plan x is subject to preregistration in benefit plan y”).
- *General procedural knowledge*: This is knowledge that can be described only by a procedure or a method (for example, a procedure to determine the “creditworthiness” of a customer).
- *Application-specific knowledge*: This is knowledge determined by the rules and regulations applicable in a specific domain (for example, computation of the cheapest airline fare between two cities).

- *Enterprise-directing knowledge*: This is a higher form of knowledge that allows an enterprise to make decisions. For example, on a companywide basis, this knowledge includes the cost of relocating and retraining employees as well as some measure of the benefit in the form of morale and loyalty of keeping employees on the job for more than n years.

Intensional knowledge is defined as knowledge that underlies and precedes the factual content of the database. Such knowledge can be fully specified *before* the database is established. Database systems exist to manage the data and the meta-data, which together constitute **extensional knowledge** or knowledge embedded in facts and instances. Knowledge systems use not only the extensional knowledge but also the intensional knowledge, possibly in the form of rules in a rule base. **Derived knowledge** is a form in which extensional knowledge and intensional knowledge are mixed. A database may store relations called father (Father_name, Child_name), mother (Mother_name, Child_name), and person (Person_name, Sex). From these three basic relations, it would be possible to define a variety of family relationships, from cousins to uncles to grandfathers, which is the job of a rule base.

Most KB systems today store intensional knowledge in the form of rules. Other representations include logical assertions, semantic networks, and frames. We refer to the inferencing and control strategy of a KB as *strategic knowledge*.

A promising new area of research is *knowledge mining*, which involves extracting previously unknown but useful information from a database (Piatesky, Shapiro, and Frawley 1991; Anwar et al. 1992). Potential areas of development include case-based reasoning (Kolodner 1993), applied as an aid to reasoning with databases; automated schema design by means of conceptual clustering (Beck et al. 1993); and explanation-based learning, applied to maintain the consistency of a database (as in the XCON system; Schimmler et al. 1991), among others. Many learning strategies may be used as new paradigms for query processing in database systems.

25.4.3 User Interfaces for Databases

Despite the current advances in user interfaces, a few areas remain to be explored, particularly for the benefit of database users:

- *Customized languages*: Casual and occasional users need languages that are easy to use and fit well with their problem domains. An account executive in an investment firm deals with a different set of terms and processing requirements than does a production supervisor in a manufacturing company. If both are using the same system, each should be given a customized interface. Currently, these are not impossible to provide, but they involve a heavy programming effort. Future systems should make it easier to generate a customized set of language constructs for each set of users.
- *Alternative paradigms for accessing databases*: The system called Spatial Data Management System (SDMS) of Computer Corporation of America (Herot 1980) is a good example of getting away from the records- and files-oriented approach to data management. SDMS simulates an environment in which data is organized in a

three-dimensional space such as an office. It bases the search for data on the principle that people are good at locating information by knowing where it is placed and by knowing some physical appearance attributes of the medium or container in which it is recorded. Such a system allows a natural “spatial exploration” through the database, encourages browsing, and uses icons as dictionary information (meta-data). This system is able to store and display illustrations and pictures from videodisks and other media. Visualization of scientific data and even of document repositories by using clever techniques is receiving a lot of attention (Korfang 1991; Chalmers 1992). In the future we might see more developments along these lines that would also incorporate other forms of information, including movies, animations, and (going a step farther) even smells and tastes!

- *Natural-language interface in multiple languages:* With faster means of communication, it is already becoming necessary to make the same database under the same DBMS available to Americans, Italians, Japanese, and others at the same time. Today’s natural-language interfaces are inefficient and inadequate. Future interfaces will demand multilingual support. There are related problems of displaying information in different scripts, allowing editing of text involving multiple languages (for example, English and Arabic, where the latter is read from right to left), storing aliases in different languages, and so on.

25.4.4 *Database Machines and Architecture*

Research on database machines has focused primarily on the efficient retrieval and storage of large quantities of data. One future direction of research on database machines would be to design a computer that integrates the problem-solving components with the data management functions. Another is to develop hardware for database management functions, such as security and integrity control, concurrency control, and transaction management, that are currently implemented by software and constitute a major source of performance overhead. Research into parallel architectures for knowledge-based systems is also needed.

25.4.5 *Interface with Programming Language Technology*

Research on query languages—query language design and query modification, optimization, and compilation—is ongoing in relation to hard application areas of CAD, office systems, and statistical, scientific, spatial, and temporal database management. Further work is needed on proving the equivalence of models and languages and on formalizing their semantics.

A new field of research called Database Programming Languages (DBPLs) is emerging (Bancilhon and Buneman 1990). These languages will minimize the “impedance mismatch” among traditional programming languages and database query languages. However, none of the proposals made so far have been demonstrated on a practical scale.

Many other fields can benefit from the application of database technology. We cannot attempt to name them all here. One promising area is Computer Supported Cooperative Work or CSCW (Kling 1991). It stands to gain by facilitating problem solving in

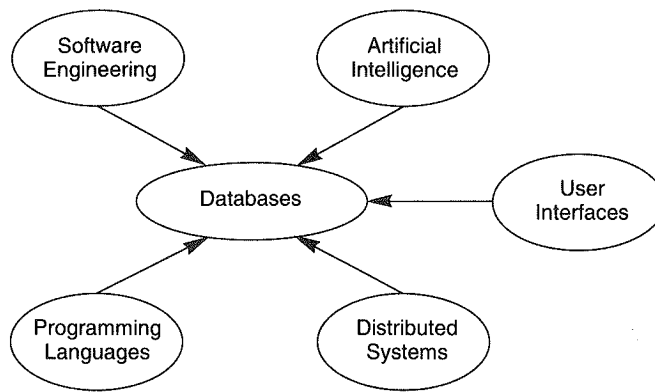


Figure 25.6 Future integration of technologies.

a group context, using principles of active database management (Chakravarthy et al. 1993).

We conclude this chapter with Figure 25.6, which shows that, in future information modeling and processing applications in the diverse areas to which databases will be applied, database technology must work hand in hand with the related technologies of software engineering, artificial intelligence, programming languages, user interfaces, and distributed systems. Users will engage in problem-solving activities with the database as a focus and with these other technologies supporting various facets of the problem-solving process.

Selected Bibliography

Much of the relevant bibliography for this chapter has been already referenced in the various sections. Here we give a few additional references. Several journals and conference proceedings may be consulted to keep track of developments in the database systems field. Among the prominent ones are *ACM Transactions on Database Systems*, *IEEE Transactions on Knowledge and Data Engineering*, *IEEE Transactions on Software Engineering* (particularly prior to 1990), *Information Systems*, and *Data and Knowledge Engineering* (North Holland). In the last couple of years, the following new journals have started: *The VLDB Journal*, *Journal of Distributed and Parallel Databases*, *Journal of Intelligent Information Systems*, and *Journal of Intelligent Cooperative Information Systems*. Several heavily refereed international conferences publish proceedings of the papers. The major conferences are the ACM SIGMOD annual conference (since 1974), the VLDB conference (since 1975), the IEEE Data Engineering Conference (since 1984), the Conference on Extending Database Technology (since 1989), and the Entity-Relationship Conference (since 1980). Because of rapidly growing interest in the emerging technologies, a new workshop called Research Issues in Data Engineering was started in 1991. It has devoted itself to interoperability and multi-databases in 1991 and 1993, to multimedia in 1992, and to active databases in 1994. Proceedings are published by IEEE. A new conference called Parallel and Distributed Information Systems has also been launched since 1991.

A series of international workshops called the Statistical Database Workshops has been held at irregular intervals starting in 1982. The International Federation of Information Processing Soci-

eties has a special interest group on databases that has been conducting Database Semantics workshops. The series of conference proceedings on Expert Database Systems (EDS 1984, 1986, 1988) has provided papers related to databases, logic, expert and knowledge-based systems, and techniques related to the integration of database and artificial intelligence technologies.

A special issue of *IEEE Transactions on Knowledge and Data Engineering* was devoted to the description of prototype database systems, some of which were mentioned in this and earlier chapters.

Reports of certain workshops summarize the results of panels invited by the National Science Foundation. One such report (Silberschatz et al. 1991) relates to the future of database systems research. Highlights from the report on scientific databases (French et al. 1990) were presented in Section 25.3.3.

Among specialized reading books, Stonebraker (1988) is a collection of research papers on database systems. Mylopoulos and Brodie (1989) has contributions on the interface of artificial intelligence and databases. Specialized books have started appearing on application areas such as geographic databases (Laurini 1992).

Another topic of current research, which we have not discussed elsewhere, involves the incorporation of incomplete and imprecise (fuzzy) information in databases. Papers that discuss incomplete information include Lipski (1979), Vassiliou (1980), Imielinski and Lipski (1981), and Liu and Sunderraman (1988). Techniques for dealing with fuzzy information are discussed in Zadeh (1983) and Zvieli (1986).

An early survey paper on back-end database machines is Maryanski (1980). Techniques for implementing main-memory databases are discussed by DeWitt et al. (1984). Applications of multimedia databases are discussed in Christodoulakis and Faloutsos (1986).

Dittrich (1986) is a survey of the object-oriented approach. Maier et al. (1986) describes the GEMSTONE /OPAL object-oriented system. The Darmstadt Database Kernel System is described in Paul et al. (1987). Navathe and Pillalamarri (1988) discusses how to make the ER model (discussed in Chapters 3 and 21) into an object-oriented model.

Many projects address databases for engineering design. Haskin and Lorie (1982) and Lorie and Plouffe (1983) discuss complex design objects in databases. Eastman (1987) discusses the facilities needed for engineering databases. Kemper et al. (1987) describes an object-oriented DBMS for engineering applications. Geometric modeling and spatial data processing are discussed by Kemper and Wallrath (1987) and Orenstein (1986). The R-tree data structure for indexing spatial data is presented in Guttman (1984). Schema evolution is discussed by Banerjee et al. (1987a), and CAD databases for VLSI are discussed in Du and Ghanta (1987).

Recently, interest has been growing in real-time databases, such as the work by Abbott and Garcia-Molina (1988). A temporal data model and query language for relational databases is described in Gadia (1988). Tsichritzis (1982) discusses forms management.