Sixth Edition

Fundamentals of Database Systems

Elmasri • Navathe

FUNDAMENTALS OF

Database Systems

SIXTH EDITION

15.6 Multivalued Dependency and Fourth Normal Form 531
15.7 Join Dependencies and Fifth Normal Form 534
15.8 Summary 535
Review Questions 536
Exercises 537
Laboratory Exercises 542
Selected Bibliography 542

chapter 16 **Relational Database Design Algorithms and Further Dependencies 543**

16.1 Further Topics in Functional Dependencies: Inference Rules, Equivalence, and Minimal Cover 545
16.2 Properties of Relational Decompositions 551
16.3 Algorithms for Relational Database Schema Design 557
16.4 About Nulls, Dangling Tuples, and Alternative Relational Designs 563
16.5 Further Discussion of Multivalued Dependencies and 4NF 567
16.6 Other Dependencies and Normal Forms 571
16.7 Summary 575
Review Questions 576
Exercises 576
Laboratory Exercises 578
Selected Bibliography 579

part 7
File Structures, Indexing, and Hashing

chapter 17 **Disk Storage, Basic File Structures, and Hashing 583**

17.1 Introduction 584
17.2 Secondary Storage Devices 587
17.3 Buffering of Blocks 593
17.4 Placing File Records on Disk 594
17.5 Operations on Files 599
17.6 Files of Unordered Records (Heap Files) 601
17.7 Files of Ordered Records (Sorted Files) 603
17.8 Hashing Techniques 606

17.9 Other Primary File Organizations 616
17.10 Parallelizing Disk Access Using RAID Technology 617
17.11 New Storage Systems 621
17.12 Summary 624
Review Questions 625
Exercises 626
Selected Bibliography 630

chapter 18 Indexing Structures for Files 631

18.1 Types of Single-Level Ordered Indexes 632
18.2 Multilevel Indexes 643
18.3 Dynamic Multilevel Indexes Using B-Trees and B^+-Trees 646
18.4 Indexes on Multiple Keys 660
18.5 Other Types of Indexes 663
18.6 Some General Issues Concerning Indexing 668
18.7 Summary 670
Review Questions 671
Exercises 672
Selected Bibliography 674

part 8
Query Processing and Optimization, and Database Tuning

chapter 19 Algorithms for Query Processing and Optimization 679

19.1 Translating SQL Queries into Relational Algebra 681
19.2 Algorithms for External Sorting 682
19.3 Algorithms for SELECT and JOIN Operations 685
19.4 Algorithms for PROJECT and Set Operations 696
19.5 Implementing Aggregate Operations and OUTER JOINs 698
19.6 Combining Operations Using Pipelining 700
19.7 Using Heuristics in Query Optimization 700
19.8 Using Selectivity and Cost Estimates in Query Optimization 710
19.9 Overview of Query Optimization in Oracle 721
19.10 Semantic Query Optimization 722
19.11 Summary 723

Review Questions 723
Exercises 724
Selected Bibliography 725

chapter 20 Physical Database Design and Tuning 727

20.1 Physical Database Design in Relational Databases 727
20.2 An Overview of Database Tuning in Relational Systems 733
20.3 Summary 739
Review Questions 739
Selected Bibliography 740

part 9
Transaction Processing, Concurrency Control, and Recovery

chapter 21 Introduction to Transaction Processing Concepts and Theory 743

21.1 Introduction to Transaction Processing 744
21.2 Transaction and System Concepts 751
21.3 Desirable Properties of Transactions 754
21.4 Characterizing Schedules Based on Recoverability 755
21.5 Characterizing Schedules Based on Serializability 759
21.6 Transaction Support in SQL 770
21.7 Summary 772
Review Questions 772
Exercises 773
Selected Bibliography 775

chapter 22 Concurrency Control Techniques 777

22.1 Two-Phase Locking Techniques for Concurrency Control 778
22.2 Concurrency Control Based on Timestamp Ordering 788
22.3 Multiversion Concurrency Control Techniques 791
22.4 Validation (Optimistic) Concurrency Control Techniques 794
22.5 Granularity of Data Items and Multiple Granularity Locking 795
22.6 Using Locks for Concurrency Control in Indexes 798
22.7 Other Concurrency Control Issues 800

22.8 Summary 802
Review Questions 803
Exercises 804
Selected Bibliography 804

chapter 23 Database Recovery Techniques 807

23.1 Recovery Concepts 808
23.2 NO-UNDO/REDO Recovery Based on Deferred Update 815
23.3 Recovery Techniques Based on Immediate Update 817
23.4 Shadow Paging 820
23.5 The ARIES Recovery Algorithm 821
23.6 Recovery in Multidatabase Systems 825
23.7 Database Backup and Recovery from Catastrophic Failures 826
23.8 Summary 827
Review Questions 828
Exercises 829
Selected Bibliography 832

part 10
Additional Database Topics: Security and Distribution

chapter 24 Database Security 835

24.1 Introduction to Database Security Issues 836
24.2 Discretionary Access Control Based on Granting and Revoking Privileges 842
24.3 Mandatory Access Control and Role-Based Access Control for Multilevel Security 847
24.4 SQL Injection 855
24.5 Introduction to Statistical Database Security 859
24.6 Introduction to Flow Control 860
24.7 Encryption and Public Key Infrastructures 862
24.8 Privacy Issues and Preservation 866
24.9 Challenges of Database Security 867
24.10 Oracle Label-Based Security 868
24.11 Summary 870

Review Questions 872
Exercises 873
Selected Bibliography 874

chapter 25 Distributed Databases 877

25.1 Distributed Database Concepts 878
25.2 Types of Distributed Database Systems 883
25.3 Distributed Database Architectures 887
25.4 Data Fragmentation, Replication, and Allocation Techniques for Distributed Database Design 894
25.5 Query Processing and Optimization in Distributed Databases 901
25.6 Overview of Transaction Management in Distributed Databases 907
25.7 Overview of Concurrency Control and Recovery in Distributed Databases 909
25.8 Distributed Catalog Management 913
25.9 Current Trends in Distributed Databases 914
25.10 Distributed Databases in Oracle 915
25.11 Summary 919
Review Questions 921
Exercises 922
Selected Bibliography 924

part 11
Advanced Database Models, Systems, and Applications

chapter 26 Enhanced Data Models for Advanced Applications 931

26.1 Active Database Concepts and Triggers 933
26.2 Temporal Database Concepts 943
26.3 Spatial Database Concepts 957
26.4 Multimedia Database Concepts 965
26.5 Introduction to Deductive Databases 970
26.6 Summary 983
Review Questions 985
Exercises 986
Selected Bibliography 989

chapter 27 **Introduction to Information Retrieval and Web Search 993**

27.1 Information Retrieval (IR) Concepts 994
27.2 Retrieval Models 1001
27.3 Types of Queries in IR Systems 1007
27.4 Text Preprocessing 1009
27.5 Inverted Indexing 1012
27.6 Evaluation Measures of Search Relevance 1014
27.7 Web Search and Analysis 1018
27.8 Trends in Information Retrieval 1028
27.9 Summary 1030
Review Questions 1031
Selected Bibliography 1033

chapter 28 **Data Mining Concepts 1035**

28.1 Overview of Data Mining Technology 1036
28.2 Association Rules 1039
28.3 Classification 1051
28.4 Clustering 1054
28.5 Approaches to Other Data Mining Problems 1057
28.6 Applications of Data Mining 1060
28.7 Commercial Data Mining Tools 1060
28.8 Summary 1063
Review Questions 1063
Exercises 1064
Selected Bibliography 1065

chapter 29 **Overview of Data Warehousing and OLAP 1067**

29.1 Introduction, Definitions, and Terminology 1067
29.2 Characteristics of Data Warehouses 1069
29.3 Data Modeling for Data Warehouses 1070
29.4 Building a Data Warehouse 1075
29.5 Typical Functionality of a Data Warehouse 1078
29.6 Data Warehouse versus Views 1079
29.7 Difficulties of Implementing Data Warehouses 1080

29.8 Summary 1081
Review Questions 1081
Selected Bibliography 1082

appendix **A** **Alternative Diagrammatic Notations for ER Models 1083**

appendix **B** **Parameters of Disks 1087**

appendix **C** **Overview of the QBE Language 1091**

C.1 Basic Retrievals in QBE 1091
C.2 Grouping, Aggregation, and Database Modification in QBE 1095

appendix **D** **Overview of the Hierarchical Data Model**
(located on the Companion Website at http://www.aw.com/elmasri)

appendix **E** **Overview of the Network Data Model**
(located on the Companion Website at http://www.aw.com/elmasri)

Selected Bibliography 1099

Index 1133

part 7

File Structures, Indexing, and Hashing

chapter 17

Disk Storage, Basic File Structures, and Hashing

Databases are stored physically as files of records, which are typically stored on magnetic disks. This chapter and the next deal with the organization of databases in storage and the techniques for accessing them efficiently using various algorithms, some of which require auxiliary data structures called *indexes*. These structures are often referred to as **physical database file structures**, and are at the physical level of the three-schema architecture described in Chapter 2. We start in Section 17.1 by introducing the concepts of computer storage hierarchies and how they are used in database systems. Section 17.2 is devoted to a description of magnetic disk storage devices and their characteristics, and we also briefly describe magnetic tape storage devices. After discussing different storage technologies, we turn our attention to the methods for physically organizing data on disks. Section 17.3 covers the technique of double buffering, which is used to speed retrieval of multiple disk blocks. In Section 17.4 we discuss various ways of formatting and storing file records on disk. Section 17.5 discusses the various types of operations that are typically applied to file records. We present three primary methods for organizing file records on disk: unordered records, in Section 17.6; ordered records, in Section 17.7; and hashed records, in Section 17.8.

Section 17.9 briefly introduces files of mixed records and other primary methods for organizing records, such as B-trees. These are particularly relevant for storage of object-oriented databases, which we discussed in Chapter 11. Section 17.10 describes RAID (Redundant Arrays of Inexpensive (or Independent) Disks)—a data storage system architecture that is commonly used in large organizations for better reliability and performance. Finally, in Section 17.11 we describe three developments in the storage systems area: storage area networks (SAN), network-

attached storage (NAS), and iSCSI (Internet SCSI—Small Computer System Interface), the latest technology, which makes storage area networks more affordable without the use of the Fiber Channel infrastructure and hence is getting very wide acceptance in industry. Section 17.12 summarizes the chapter. In Chapter 18 we discuss techniques for creating auxiliary data structures, called indexes, which speed up the search for and retrieval of records. These techniques involve storage of auxiliary data, called index files, in addition to the file records themselves.

Chapters 17 and 18 may be browsed through or even omitted by readers who have already studied file organizations and indexing in a separate course. The material covered here, in particular Sections 17.1 through 17.8, is necessary for understanding Chapters 19 and 20, which deal with query processing and optimization, and database tuning for improving performance of queries.

17.1 Introduction

The collection of data that makes up a computerized database must be stored physically 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's *central processing unit* (CPU), such as the computer's main memory and smaller but faster cache memories. Primary storage usually provides fast access to data but is of limited storage capacity. Although main memory capacities have been growing rapidly in recent years, they are still more expensive and have less storage capacity than secondary and tertiary storage devices.
- **Secondary and tertiary storage.** This category includes magnetic disks, optical disks (CD-ROMs, DVDs, and other similar storage media), and tapes. Hard-disk drives are classified as secondary storage, whereas removable media such as optical disks and tapes are considered tertiary storage. These devices usually have a larger capacity, cost less, and provide slower access to data than do primary storage devices. Data in secondary or tertiary storage cannot be processed directly by the CPU; first it must be copied into primary storage and then processed by the CPU.

We first give an overview of the various storage devices used for primary and secondary storage in Section 17.1.1 and then discuss how databases are typically handled in the storage hierarchy in Section 17.1.2.

17.1.1 Memory Hierarchies and Storage Devices

In a modern computer system, data resides and is transported throughout a hierarchy of storage media. The highest-speed memory is the most expensive and is therefore available with the least capacity. The lowest-speed memory is offline tape storage, which is essentially available in indefinite storage capacity.

At the *primary storage level,* the memory hierarchy includes at the most expensive end, **cache memory**, which is a static RAM (Random Access Memory). Cache memory is typically used by the CPU to speed up execution of program instructions using techniques such as prefetching and pipelining. The next level of primary storage is DRAM (Dynamic RAM), which provides the main work area for the CPU for keeping program instructions and data. It is popularly called **main memory**. The advantage of DRAM is its low cost, which continues to decrease; the drawback is its volatility[1] and lower speed compared with static RAM. At the *secondary and tertiary storage level,* the hierarchy includes magnetic disks, as well as **mass storage** in the form of CD-ROM (Compact Disk–Read-Only Memory) and DVD (Digital Video Disk or Digital Versatile Disk) devices, and finally tapes at the least expensive end of the hierarchy. The **storage capacity** is measured in kilobytes (Kbyte or 1000 bytes), megabytes (MB or 1 million bytes), gigabytes (GB or 1 billion bytes), and even terabytes (1000 GB). The word petabyte (1000 terabytes or 10**15 bytes) is now becoming relevant in the context of very large repositories of data in physics, astronomy, earth sciences, and other scientific applications.

Programs reside and execute in DRAM. Generally, large permanent databases reside on secondary storage, (magnetic disks), and portions of the database are read into and written from buffers in main memory as needed. Nowadays, personal computers and workstations have large main memories of hundreds of megabytes of RAM and DRAM, so it is becoming possible to load a large part of the database into main memory. Eight to 16 GB of main memory on a single server is becoming commonplace. In some cases, entire databases can be kept in main memory (with a backup copy on magnetic disk), leading to **main memory databases**; these are particularly useful in real-time applications that require extremely fast response times. An example is telephone switching applications, which store databases that contain routing and line information in main memory.

Between DRAM and magnetic disk storage, another form of memory, **flash memory**, is becoming common, particularly because it is nonvolatile. Flash memories are high-density, high-performance memories using EEPROM (Electrically Erasable Programmable Read-Only Memory) technology. The advantage of flash memory is the fast access speed; the disadvantage is that an entire block must be erased and written over simultaneously. Flash memory cards are appearing as the data storage medium in appliances with capacities ranging from a few megabytes to a few gigabytes. These are appearing in cameras, MP3 players, cell phones, PDAs, and so on. USB (Universal Serial Bus) flash drives have become the most portable medium for carrying data between personal computers; they have a flash memory storage device integrated with a USB interface.

CD-ROM (Compact Disk – Read Only Memory) disks store data optically and are read by a laser. CD-ROMs contain prerecorded data that cannot be overwritten. WORM (Write-Once-Read-Many) disks are a form of optical storage used for

[1]Volatile memory typically loses its contents in case of a power outage, whereas nonvolatile memory does not.

archiving data; they allow data to be written once and read any number of times without the possibility of erasing. They hold about half a gigabyte of data per disk and last much longer than magnetic disks.[2] **Optical jukebox memories** use an array of CD-ROM platters, which are loaded onto drives on demand. Although optical jukeboxes have capacities in the hundreds of gigabytes, their retrieval times are in the hundreds of milliseconds, quite a bit slower than magnetic disks. This type of storage is continuing to decline because of the rapid decrease in cost and increase in capacities of magnetic disks. The DVD is another standard for optical disks allowing 4.5 to 15 GB of storage per disk. Most personal computer disk drives now read CD-ROM and DVD disks. Typically, drives are CD-R (Compact Disk Recordable) that can create CD-ROMs and audio CDs (Compact Disks), as well as record on DVDs.

Finally, **magnetic tapes** are used for archiving and backup storage of data. **Tape jukeboxes**—which contain a bank of tapes that are catalogued and can be automatically loaded onto tape drives—are becoming popular as **tertiary storage** to hold terabytes of data. For example, NASA's EOS (Earth Observation Satellite) system stores archived databases in this fashion.

Many large organizations are already finding it normal to have terabyte-sized databases. The term **very large database** can no longer be precisely defined because disk storage capacities are on the rise and costs are declining. Very soon the term may be reserved for databases containing tens of terabytes.

17.1.2 Storage of Databases

Databases typically store large amounts of data that must persist over long periods of time, and hence is often referred to as **persistent data**. Parts of this data are accessed and processed repeatedly during this period. This contrasts with the notion of **transient data** that persist for only a limited time during program execution. Most databases are stored permanently (or *persistently*) 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 secondary storage than for primary storage.

Some of the newer technologies—such as optical disks, DVDs, and tape jukeboxes—are likely to provide viable alternatives to the use of magnetic disks. In the future, databases may therefore reside at different levels of the memory hierarchy from those described in Section 17.1.1. However, it is anticipated that magnetic

[2]Their rotational speeds are lower (around 400 rpm), giving higher latency delays and low transfer rates (around 100 to 200 KB/second).

disks will continue to be the primary medium of choice for large databases for years to come. Hence, 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 databases 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 **offline**; that is, some intervention by an operator—or an automatic loading device—to load a tape is needed before the data becomes available. In contrast, disks are **online** devices that can be accessed directly at any time.

The techniques used to store large amounts of structured data on disk 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 has several options available for organizing the data. The process of **physical database design** involves choosing the particular data organization techniques that best suit the given application requirements from among the 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 when they are needed.

There are several **primary file organizations**, which determine how the file records are *physically placed* on the disk, *and hence how the records can be accessed.* A *heap file* (or *unordered file*) places the records on disk in no particular order by appending new records at the end of the file, whereas a *sorted file* (or *sequential file*) keeps the records ordered by the value of a particular field (called the *sort key*). A *hashed file* uses a hash function applied to a particular field (called the *hash key*) to determine a record's placement on disk. Other primary file organizations, such as *B-trees,* use tree structures. We discuss primary file organizations in Sections 17.6 through 17.9. A **secondary organization** or **auxiliary access structure** allows efficient access to file records based on *alternate fields* than those that have been used for the primary file organization. Most of these exist as indexes and will be discussed in Chapter 18.

17.2 Secondary Storage Devices

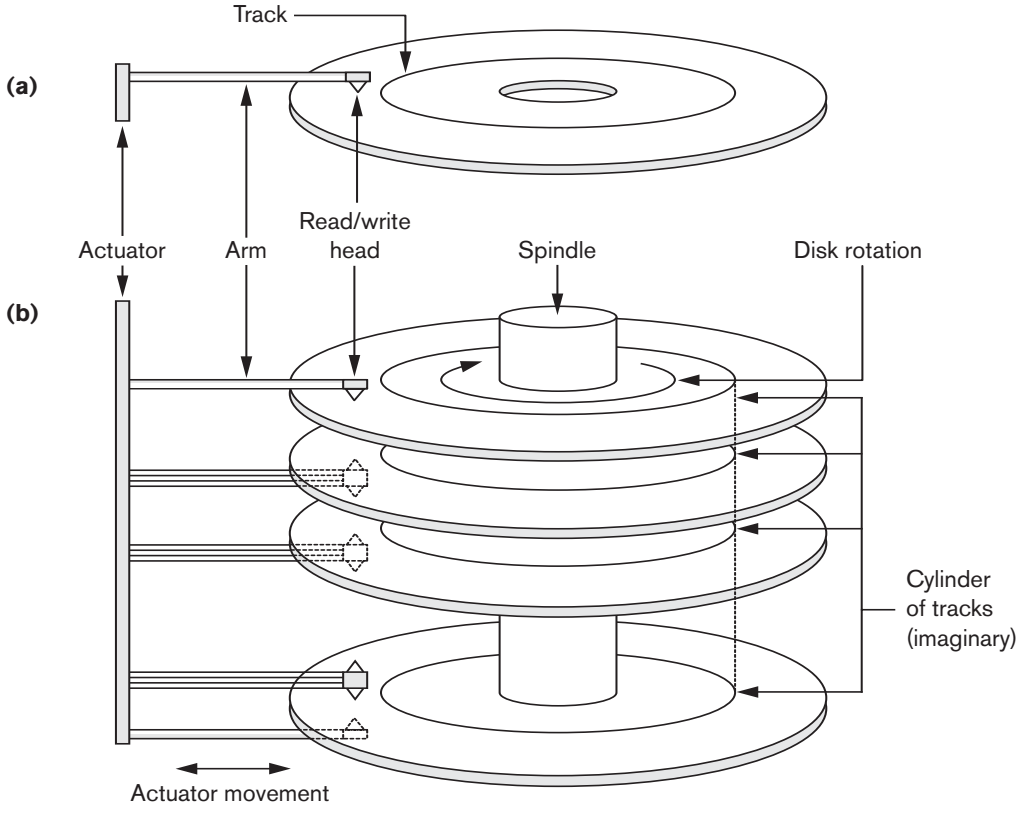
In this section we describe some characteristics of magnetic disk and magnetic tape storage devices. Readers who have already studied these devices may simply browse through this section.

17.2.1 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. Small floppy disks used with microcomputers typically hold from 400 KB to 1.5 MB; they are rapidly going out of circulation. Hard disks for personal computers typically hold from several hundred MB up to tens of GB; and large disk packs used with servers and mainframes have capacities of hundreds of GB. Disk capacities continue to grow as technology improves.

Whatever their capacity, all disks are made of magnetic material shaped as a thin circular disk, as shown in Figure 17.1(a), and protected by a plastic or acrylic cover.

Figure 17.1
(a) A single-sided disk with read/write hardware.
(b) A disk pack with read/write hardware.



A disk is **single-sided** if it stores information on one of its surfaces only and **double-sided** if both surfaces are used. To increase storage capacity, disks are assembled into a **disk pack**, as shown in Figure 17.1(b), which may include many disks and therefore many surfaces. Information is stored on a disk surface in concentric circles of *small width,*[3] each having a distinct diameter. Each circle is called a **track**. In disk packs, tracks with the same diameter on the various surfaces are 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 one cylinder can be retrieved much faster than if it were distributed among different cylinders.

The number of tracks on a disk ranges from a few hundred to a few thousand, and the capacity of each track typically ranges from tens of Kbytes to 150 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. One type of sector organization, as shown in Figure 17.2(a), calls a portion of a track that subtends a fixed angle at the center a sector. Several other sector organizations are possible, one of which is to have the sectors subtend smaller angles at the center as one moves away, thus maintaining a uniform density of recording, as shown in Figure 17.2(b). A technique called ZBR (Zone Bit Recording) allows a range of cylinders to have the same number of sectors per arc. For example, cylinders 0–99 may have one sector per track, 100–199 may have two per track, and so on. Not all disks have their tracks divided into sectors.

The division of a track into equal-sized **disk 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 8192 bytes. A disk with hard-coded sectors often has the sectors subdivided into blocks during initialization. Blocks are separated by fixed-size **interblock gaps**, which include specially coded control information written during disk initialization. This information is used to determine which block on the track follows each

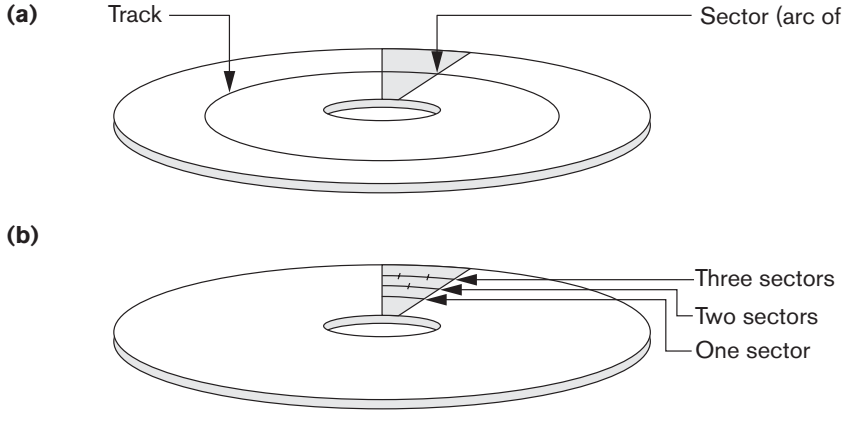


Figure 17.2
Different sector organizations on disk. (a) Sectors subtending a fixed angle. (b) Sectors maintaining a uniform recording density.

[3]In some disks, the circles are now connected into a kind of continuous spiral.

interblock gap. Table 17.1 illustrates the specifications of typical disks used on large servers in industry. The 10K and 15K prefixes on disk names refer to the rotational speeds in rpm (revolutions per minute).

There is continuous improvement in the storage capacity and transfer rates associated with disks; they are also progressively getting cheaper—currently costing only a fraction of a dollar per megabyte of disk storage. Costs are going down so rapidly that costs as low 0.025 cent/MB—which translates to $0.25/GB and $250/TB—are already here.

A disk is a *random access* addressable device. Transfer of data between main memory and disk takes place in units of disk blocks. The **hardware address** of a block—a combination of a cylinder number, track number (surface number within the cylinder on which the track is located), and block number (within the track) is supplied to the disk I/O (input/output) hardware. In many modern disk drives, a single number called LBA (Logical Block Address), which is a number between 0 and n (assuming the total capacity of the disk is $n + 1$ blocks), is mapped automatically to the right block by the disk drive controller. The address of a **buffer**—a contiguous

Table 17.1 Specifications of Typical High-End Cheetah Disks from Seagate

Description	Cheetah 15K.6	Cheetah NS 10K
Model Number	ST3450856SS/FC	ST3400755FC
Height	25.4 mm	26.11 mm
Width	101.6 mm	101.85 mm
Length	146.05 mm	147 mm
Weight	0.709 kg	0.771 kg
Capacity		
Formatted Capacity	450 Gbytes	400 Gbytes
Configuration		
Number of disks (physical)	4	4
Number of heads (physical)	8	8
Performance		
Transfer Rates		
Internal Transfer Rate (min)	1051 Mb/sec	
Internal Transfer Rate (max)	2225 Mb/sec	1211 Mb/sec
Mean Time Between Failure (MTBF)		1.4 M hours
Seek Times		
Avg. Seek Time (Read)	3.4 ms (typical)	3.9 ms (typical)
Avg. Seek Time (Write)	3.9 ms (typical)	4.2 ms (typical)
Track-to-track, Seek, Read	0.2 ms (typical)	0.35 ms (typical)
Track-to-track, Seek, Write	0.4 ms (typical)	0.35 ms (typical)
Average Latency	2 ms	2.98 msec

Courtesy Seagate Technology

reserved area in main storage that holds one disk block—is also provided. For a **read** command, the disk block 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, as shown in Figure 17.1(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 (typically ranging between 5,400 and 15,000 rpm). 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.

A **disk controller**, typically embedded in the disk drive, controls the disk drive and interfaces it to the computer system. One of the standard interfaces used today for disk drives on PCs and workstations is called **SCSI** (Small Computer System Interface). The controller accepts high-level I/O commands and takes appropriate action to position the arm and causes the read/write action to take place. To transfer a disk block, given its address, the disk controller must first mechanically position the read/write head on the correct track. The time required to do this is called the **seek time**. Typical seek times are 5 to 10 msec on desktops and 3 to 8 msecs on servers. 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. It depends on the rpm of the disk. For example, at 15,000 rpm, the time per rotation is 4 msec and the average rotational delay is the time per half revolution, or 2 msec. At 10,000 rpm the average rotational delay increases to 3 msec. 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 9 to 60 msec. For contiguous blocks, locating the first block takes from 9 to 60 msec, but transferring subsequent blocks may take only 0.4 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 with 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 18 attempt to *minimize the number of block transfers* needed to locate and transfer the required data from disk to main memory. Placing "related information" on contiguous blocks is the basic goal of any storage organization on disk.

17.2.2 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 nth block on tape, first we must scan the preceding $n - 1$ blocks. Data is stored on reels of high-capacity magnetic tape, somewhat similar to audiotapes or videotapes. A tape drive is required to read the data from or write the data to a **tape reel**. Usually, each group of bits that forms 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[4] of 0.6 inch corresponds to 960 to 3750 bytes of wasted storage space. It is customary to group many records together in one block for better space utilization.

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, the tape is mounted and then scanned 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 online data, except for some specialized applications. However, tapes serve a very important function—**backing up** the database. One reason for backup is to keep copies of disk files in case the data is lost due to a disk crash, which can happen if the disk read/write head touches the disk surface because of mechanical malfunction. For this reason, disk files are copied periodically to tape. For many online critical applications, such as airline reservation systems, to avoid any downtime, mirrored systems are used to keep three sets of identical disks—two in online operation and one

[4]Called *interrecord gaps* in tape terminology.

as backup. Here, offline disks become a backup device. The three are rotated so that they can be switched in case there is a failure on one of the live disk drives. Tapes can also be used to store excessively large database files. Database files that are seldom used or are outdated but required for historical record keeping can be **archived** on tape. Originally, half-inch reel tape drives were used for data storage employing the so-called 9 track tapes. Later, smaller 8-mm magnetic tapes (similar to those used in camcorders) that can store up to 50 GB, as well as 4-mm helical scan data cartridges and writable CDs and DVDs, became popular media for backing up data files from PCs and workstations. They are also used for storing images and system libraries.

Backing up enterprise databases so that no transaction information is lost is a major undertaking. Currently, tape libraries with slots for several hundred cartridges are used with Digital and Superdigital Linear Tapes (DLTs and SDLTs) having capacities in hundreds of gigabytes that record data on linear tracks. Robotic arms are used to write on multiple cartridges in parallel using multiple tape drives with automatic labeling software to identify the backup cartridges. An example of a giant library is the SL8500 model of Sun Storage Technology that can store up to 70 petabytes (petabyte = 1000 TB) of data using up to 448 drives with a maximum throughput rate of 193.2 TB/hour. We defer the discussion of disk storage technology called RAID, and of storage area networks, network-attached storage, and iSCSI storage systems to the end of the chapter.

17.3 Buffering of Blocks

When several blocks need to be transferred from disk to main memory and all the block addresses are known, 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 because an independent disk I/O processor (controller) exists that, once started, can proceed to transfer a data block between memory and disk independent of and in parallel to CPU processing.

Figure 17.3 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 **parallel** fashion. When a single CPU controls multiple processes, parallel 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 parallel fashion, either because a separate disk I/O processor is available or because multiple CPU processors exist.

Figure 17.4 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 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 read a continuous stream of blocks from disk to memory. Double buffering permits continuous reading or writing of data on consecutive disk blocks, which eliminates the seek time and rotational delay

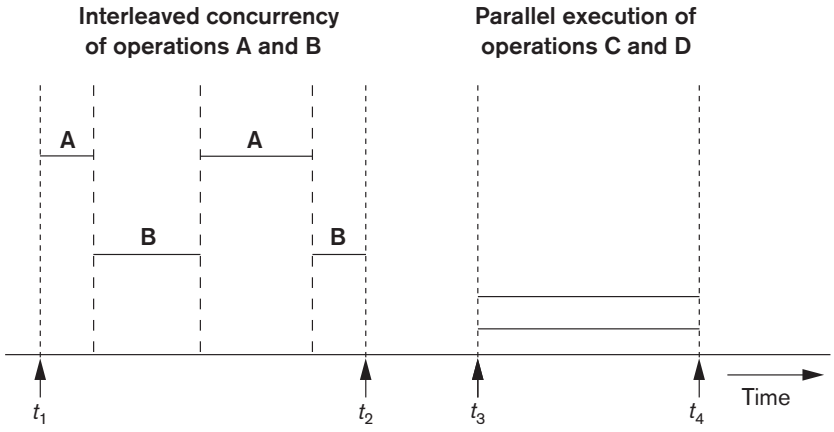


Figure 17.3
Interleaved concurrency versus parallel execution.

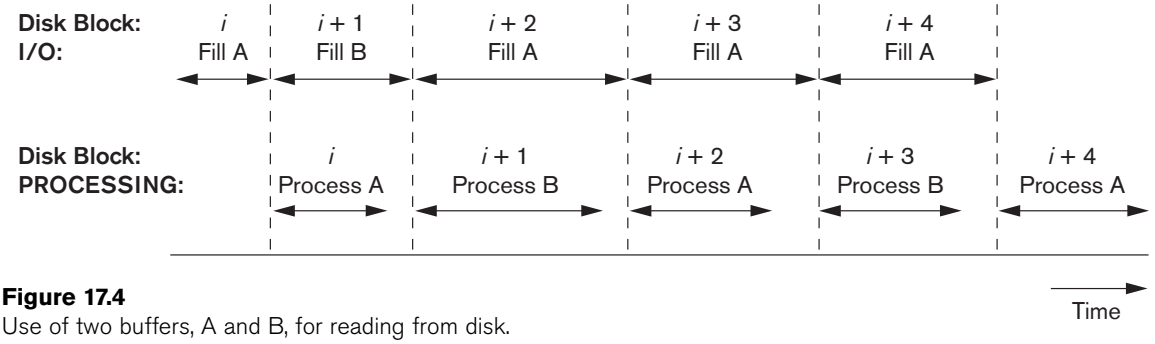


Figure 17.4
Use of two buffers, A and B, for reading from disk.

for all but the first block transfer. Moreover, data is kept ready for processing, thus reducing the waiting time in the programs.

17.4 Placing File Records on Disk

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

17.4.1 Records and 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 corre-

sponding data types constitutes a **record type** or **record format** definition. A **data type**, associated with each field, specifies the types 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 floating point), 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 10 bytes (assuming a format of YYYY-MM-DD), 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 the C programming language notation—as the following structure:

```
struct employee{
    char name[30];
    char ssn[9];
    int salary;
    int job_code;
    char department[20];
} ;
```

In some 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 **BLOB**s (binary large objects). A BLOB data item is typically stored separately from its record in a pool of disk blocks, and a pointer to the BLOB is included in the record.

17.4.2 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 *clustered* (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 17.5(a) have a record size of 71 bytes. Every record has the same fields, and field lengths are fixed, so the system 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 repeating field, we could allocate as many spaces in each record as the *maximum possible number of occurrences* of the field. In either case, space is wasted when certain records do not have values for all the physical spaces provided in each record. Now we consider other options for formatting records of a file of variable-length records.

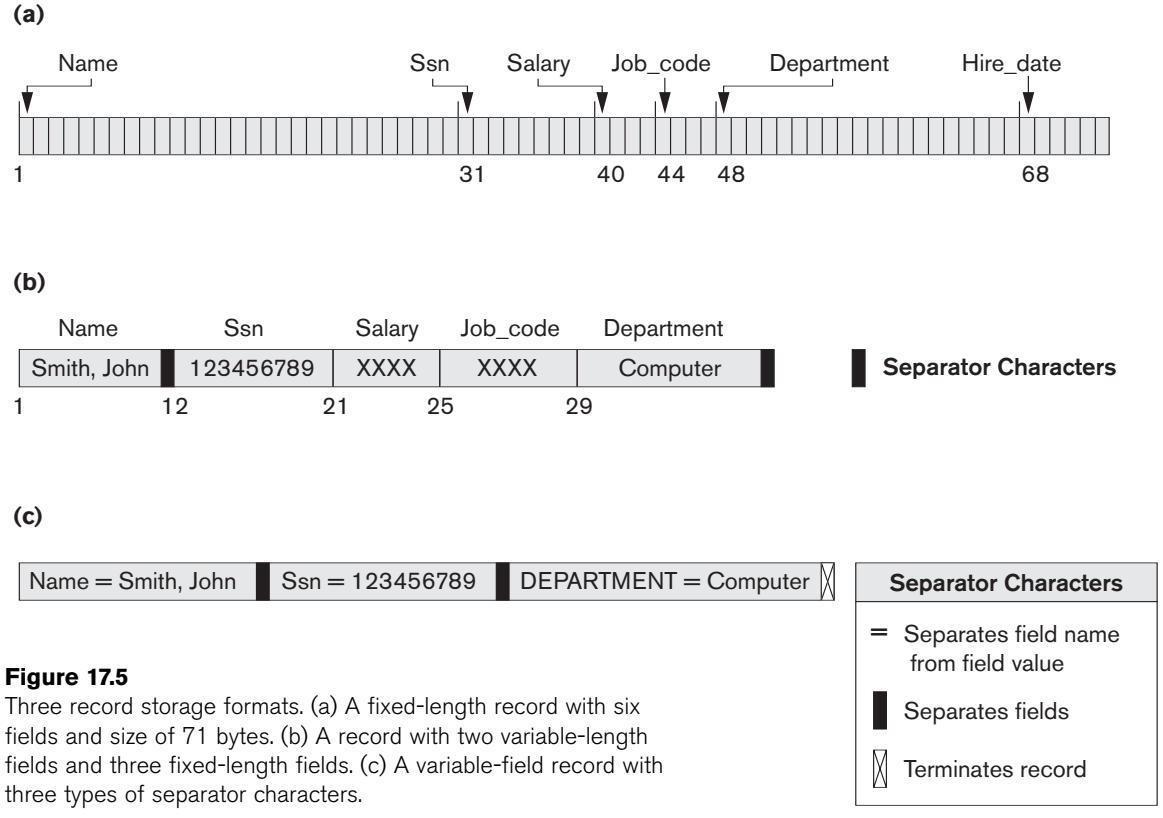


Figure 17.5
Three record storage formats. (a) A fixed-length record with six fields and size of 71 bytes. (b) A record with two variable-length fields and three fixed-length fields. (c) A variable-field record with three types of separator characters.

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 (such as ? or % or $)—which do not appear in any field value—to terminate variable-length fields, as shown in Figure 17.5(b), or we can store the length in bytes of the field in the record, preceding the field value.

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 17.5(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** code—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—which are usually part of the file system and hence hidden from the typical programmers—need to be more complex than those for fixed-length records, where the starting position and size of each field are known and fixed.[5]

17.4.3 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, although some files may have unusually large records 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 down* the number x to an 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. 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 $B > R$ because it makes

[5]Other schemes are also possible for representing variable-length records.

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 is large, it is advantageous to use spanning to reduce the lost space in each block. Figure 17.6 illustrates spanned versus unspanned organization.

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.

17.4.4 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. Clusters are sometimes called **file 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.

17.4.5 File Headers

A **file header** or **file descriptor** contains information about a file that is needed by the system 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 the 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, using 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

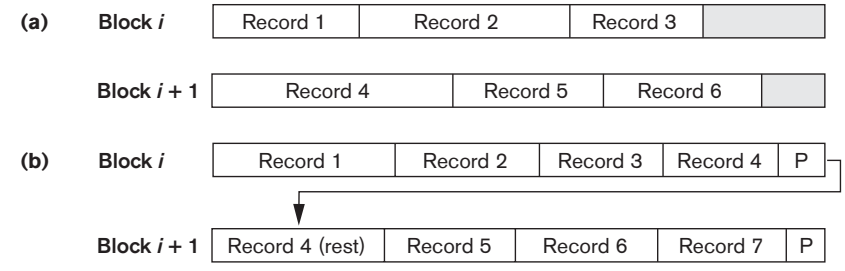


Figure 17.6
Types of record organization. (a) Unspanned. (b) Spanned.

the file blocks. Each file block is copied into a buffer and searched until 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.

17.5 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** (or **filtering condition**), which specifies criteria that the desired record or records must satisfy.

Consider an EMPLOYEE file with fields Name, Ssn, Salary, Job_code, 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 files 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, the *first* record—with respect to the physical sequence of file records—is initially located and designated the **current record**. Subsequent search operations commence from this record and locate the *next* record in the file that satisfies 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 records by using these commands, so we sometimes refer to **program variables** in the following descriptions:

- **Open.** Prepares the file for reading or writing. Allocates appropriate buffers (typically at least two) to hold file blocks from disk, and retrieves the file header. Sets the file pointer to the beginning of the file.
- **Reset.** Sets the file pointer of an open file to the beginning of the file.
- **Find (or Locate).** Searches for the first record that satisfies a search condition. Transfers the block containing that record into a main memory buffer (if it is not already there). The file pointer points to the record in the buffer

and it becomes the *current record.* Sometimes, different verbs are used to indicate whether the located record is to be retrieved or updated.

- **Read (or Get).** Copies the current record from the buffer to a program variable 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. Various forms of FindNext (for example, Find Next record within a current parent record, Find Next record of a given type, or Find Next record where a complex condition is met) are available in legacy DBMSs based on the hierarchical and network models.
- **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.
- **Close.** Completes the file access by releasing the buffers and performing any other needed cleanup operations.

The preceding (except for Open and Close) are called **record-at-a-time** operations because each operation applies to a single record. It is possible to streamline the operations Find, FindNext, and Read into a single operation, Scan, whose description is as follows:

- **Scan.** If the file has just been opened or reset, *Scan* returns the first record; otherwise it returns the next record. If a condition is specified with the operation, the returned record is the first or next record satisfying the condition.

In database systems, additional **set-at-a-time** higher-level operations may be applied to a file. Examples of these are as follows:

- **FindAll.** Locates *all* the records in the file that satisfy a search condition.
- **Find (or Locate) *n*.** Searches for the first record that satisfies a search condition and then continues to locate the next $n - 1$ records satisfying the same condition. Transfers the blocks containing the n records to the main memory buffer (if not already there).
- **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.

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, provides a group of operations—such as those listed earlier—that can be applied to a file. In general, it is possible to apply several 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 18).

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 **dynamic** 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, as shown in Figure 17.5(a), which stores the records for current employees in a company. We expect to insert records (when employees are hired), delete records (when employees leave the company), and modify records (for example, when an employee's salary or job 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 18). Suppose that a second application uses the file to generate employees' paychecks and requires that paychecks are grouped by department. For this application, it is best to order employee records by department and then by name within each department. The clustering of records into blocks and the organization of blocks on cylinders would now be different than before. However, this arrangement conflicts with ordering the records by Ssn values. If both applications are important, the designer should choose an organization that allows both operations to be done efficiently. Unfortunately, in many cases a single organization does not allow all needed operations on a file to be implemented efficiently. This requires that a compromise must be chosen that takes into account the expected importance and mix of retrieval and update operations.

In the following sections and in Chapter 18, we discuss methods for organizing records of a file on disk. Several general techniques, such as ordering, hashing, and indexing, are used to create access methods. Additionally, various general techniques for handling insertions and deletions work with many file organizations.

17.6 Files of Unordered Records (Heap Files)

In this simplest and most basic type of organization, records are placed in the file in the order in which they are inserted, so new records are inserted at the end of the

file. Such an organization is called a **heap** or **pile file**.[6] This organization is often used with additional access paths, such as the secondary indexes discussed in Chapter 18. 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 its block, copy the block into a buffer, delete the record from the buffer, and finally **rewrite the block** back to the disk. This leaves 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 for 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 it may be used with 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 techniques for **external sorting** are used (see Chapter 19).

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 ith record of the file is located in block $\lfloor (i/bfr) \rfloor$ and is the $(i \bmod bfr)$th record in that block. Such a file is often called a **relative** or **direct file** because records can easily be accessed directly by their relative positions. 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 18.

[6]Sometimes this organization is called a **sequential file**.

17.7 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.[7] 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 called the **ordering key** for the file. Figure 17.7 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 key values becomes extremely efficient because no sorting is required. Second, finding the next record from the current one in order of the ordering key 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. Ordered files are blocked and stored on contiguous cylinders to minimize the seek time.

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 17.1. 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.

Algorithm 17.1. Binary Search on an Ordering Key of a Disk File

```
l ← 1; u ← b; (* b is the number of file blocks *)
while (u ≥ l ) do
    begin i ← (l + u) div 2;
    read block i of the file into the buffer;
    if K < (ordering key field value of the first record in block i )
        then u ← i – 1
    else if K > (ordering key field value of the last record in block i )
        then l ← 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 >, <, ≥, and ≤ on the ordering field is quite efficient, since the physical ordering of records means that all records

[7]The term *sequential file* has also been used to refer to unordered files, although it is more appropriate for ordered files.

satisfying the condition are contiguous in the file. For example, referring to Figure 17.7, if the search criterion is (Name < 'G')—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 'G'.

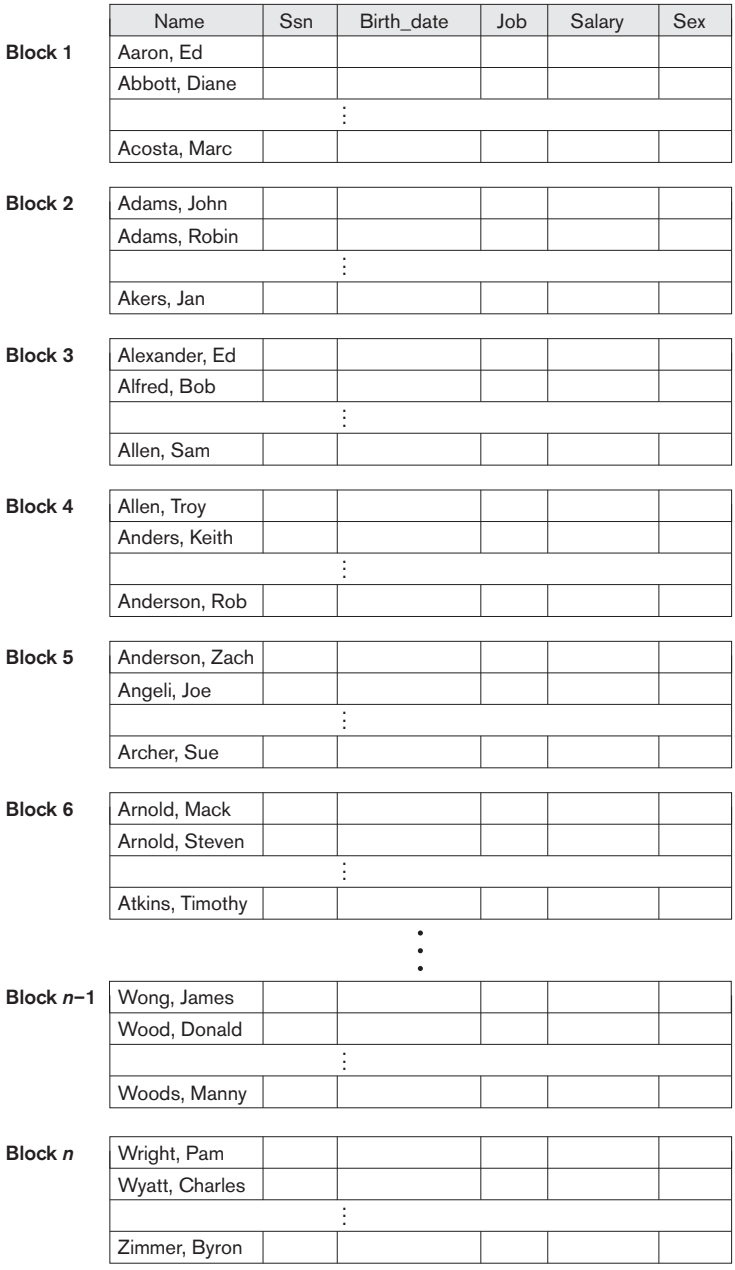


Figure 17.7
Some blocks of an ordered (sequential) file of EMPLOYEE records with Name as the ordering key field.

Ordering does not provide any advantages for random or ordered access of the records based on values of the other *nonordering fields* 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—in a different order—of the file.

Inserting and deleting records are expensive operations for an ordered file because the records must remain physically ordered. To insert a 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 deletion markers and periodic reorganization are used.

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 sorted and 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 ordering 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. This 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, first we can reorganize the file, and then read its blocks sequentially. To reorganize the file, first we sort the records in the overflow file, and then merge them with the master file. The records marked for deletion are removed during the reorganization.

Table 17.2 summarizes the average access time in block accesses to find a specific record in a file with b blocks.

Ordered files are rarely used in database applications unless an additional access path, called a **primary index**, is used; this results in an **indexed-sequential file**. This

Table 17.2 Average Access Times for a File of *b* Blocks under Basic File Organizations

Type of Organization	Access/Search Method	Average Blocks to Access a Specific Record
Heap (unordered)	Sequential scan (linear search)	$b/2$
Ordered	Sequential scan	$b/2$
Ordered	Binary search	$\log_2 b$

further improves the random access time on the ordering key field. (We discuss indexes in Chapter 18.) If the ordering attribute is not a key, the file is called a **clustered file**.

17.8 Hashing Techniques

Another type of primary file organization is based on hashing, which provides very fast access to records under certain search conditions. This organization is usually called a **hash file**.[8] The search condition must be an equality condition on a single field, called the **hash field**. In most cases, 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**, which 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 search structure within a program whenever a group of records is accessed exclusively by using the value of one field. We describe the use of hashing for internal files in Section 17.8.1; then we show how it is modified to store external files on disk in Section 17.8.2. In Section 17.8.3 we discuss techniques for extending hashing to dynamically growing files.

17.8.1 Internal Hashing

For internal files, hashing is typically implemented as a **hash table** through the use of an array of records. Suppose that the array index range is from 0 to $M - 1$, as shown in Figure 17.8(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 $h(K) = K$ **mod** 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.

[8]A hash file has also been called a *direct file*.

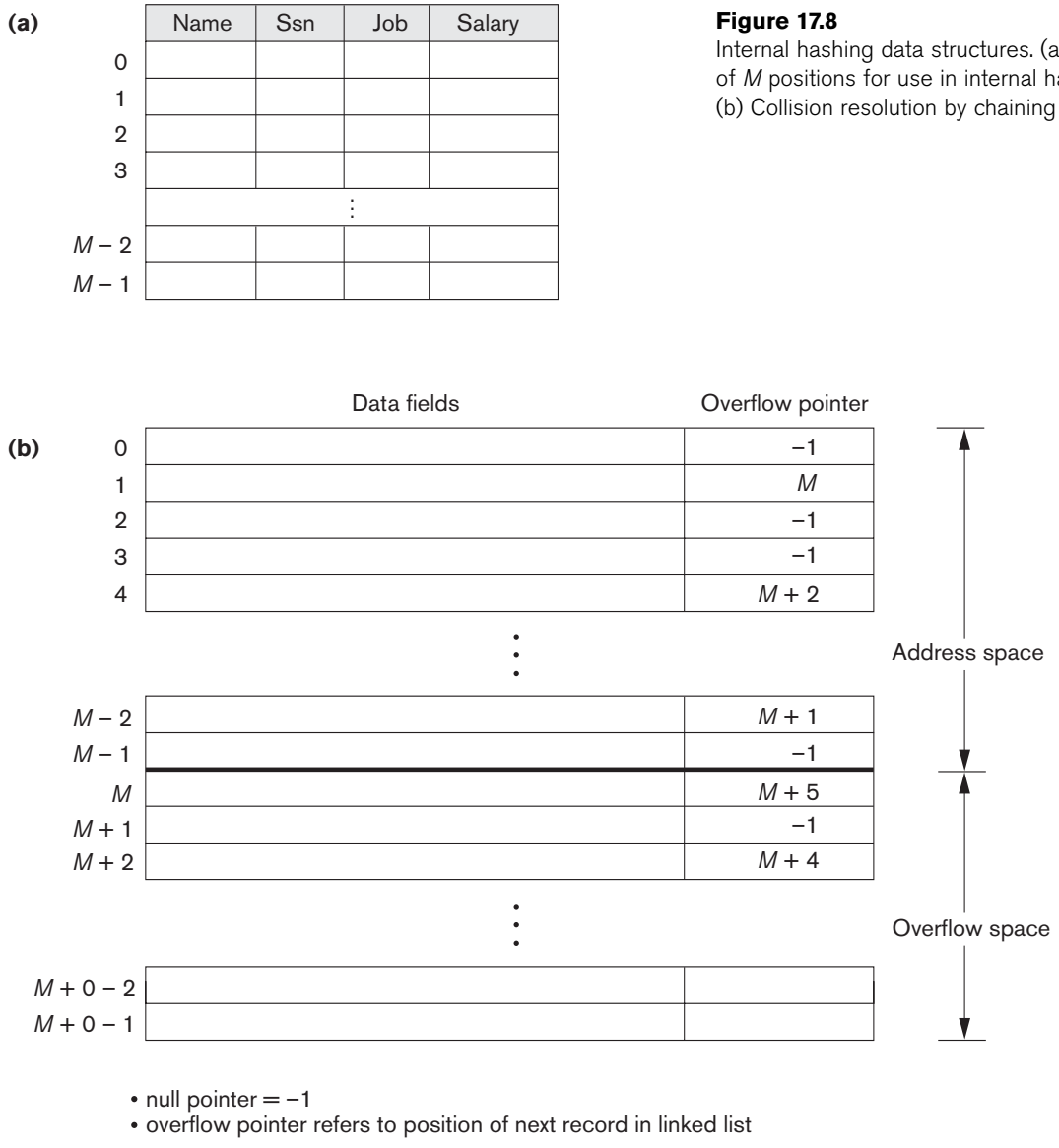


Figure 17.8
Internal hashing data structures. (a) Array of M positions for use in internal hashing. (b) Collision resolution by chaining records.

Noninteger hash field values can be transformed into integers before the mod function is applied. For character strings, the numeric (ASCII) codes associated with characters can be used in the transformation—for example, by multiplying those code values. For a hash field whose data type is a string of 20 characters, Algorithm 17.2(a) can be used to calculate the hash address. We assume that the code function returns the numeric code of a character and that we are given a hash field value K of type *K: array* [1..20] *of char* (in Pascal) or *char K*[20] (in C).

Algorithm 17.2. Two simple hashing algorithms: (a) Applying the mod hash function to a character string K. (b) Collision resolution by open addressing.

(a) *temp* ← 1;
for *i* ← 1 to 20 do *temp* ← *temp* * code(*K*[*i*]) mod *M* ;
hash_address ← *temp* mod *M*;

(b) *i* ← *hash_address*(*K*); *a* ← *i*;
if location *i* is occupied
then **begin** *i* ← (*i* + 1) mod *M*;
while (*i* ≠ *a*) and location *i* is occupied
do *i* ← (*i* + 1) mod *M*;
if (*i* = *a*) then all positions are full
else *new_hash_address* ← *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 (for example, with an address space from 0 to 999 to store 1,000 keys, a 6-digit key 235469 may be folded and stored at the address: (235+964) mod 1000 = 199). Another technique involves picking some digits of the hash field value—for instance, the third, fifth, and eighth digits—to form the hash address (for example, storing 1,000 employees with Social Security numbers of 10 digits into a hash file with 1,000 positions would give the Social Security number 301-67-8923 a hash value of 172 by this hash function).[9] 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 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 17.2(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. Additionally, 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.

[9]A detailed discussion of hashing functions is outside the scope of our presentation.

A linked list of overflow records for each hash address is thus maintained, as shown in Figure 17.8(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 algorithms 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 percent 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.

17.8.2 External Hashing for Disk Files

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 disk 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 17.9.

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 a pointer is maintained in each bucket to a linked list of overflow records for the bucket, as shown in Figure 17.10. The pointers in the linked list should be **record pointers**, which include both a block address and a relative record position within the block.

Hashing provides the fastest possible access for retrieving an arbitrary record given the value of its hash field. Although most good hash functions do not maintain records in order of hash field values, some functions—called **order preserving**—do. A simple example of an order preserving hash function is to take the leftmost three digits of an invoice number field that yields a bucket address as the hash address and keep the records sorted by invoice number within each bucket. Another

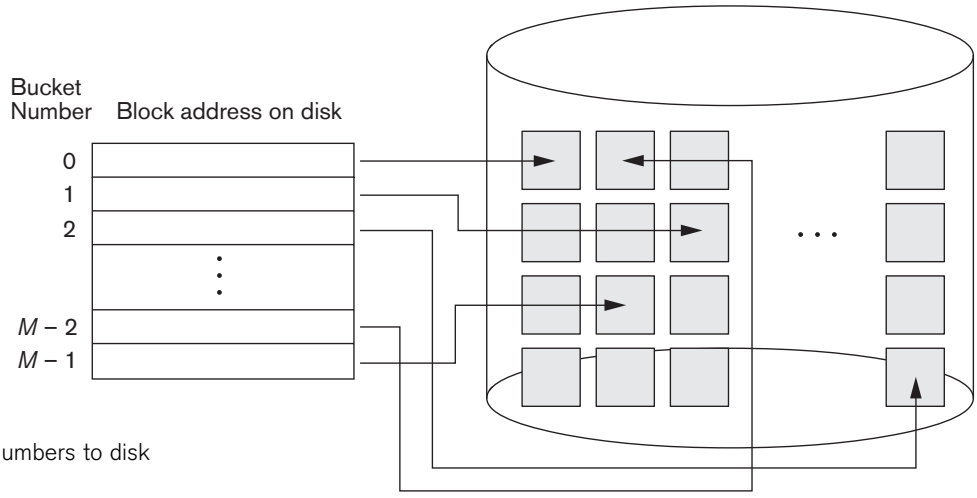


Figure 17.9
Matching bucket numbers to disk block addresses.

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.

The hashing scheme described so far is called **static hashing** because a fixed number of buckets M is allocated. This can be a serious drawback for dynamic files. 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 fewer 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 M allocated and then use a new hashing function (based on the new value of M) to redistribute the records. These reorganizations can be quite time-consuming for large files. Newer dynamic file organizations based on hashing allow the number of buckets to vary dynamically with only localized reorganization (see Section 17.8.3).

When using 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 done easily by maintaining a linked list of unused overflow locations.

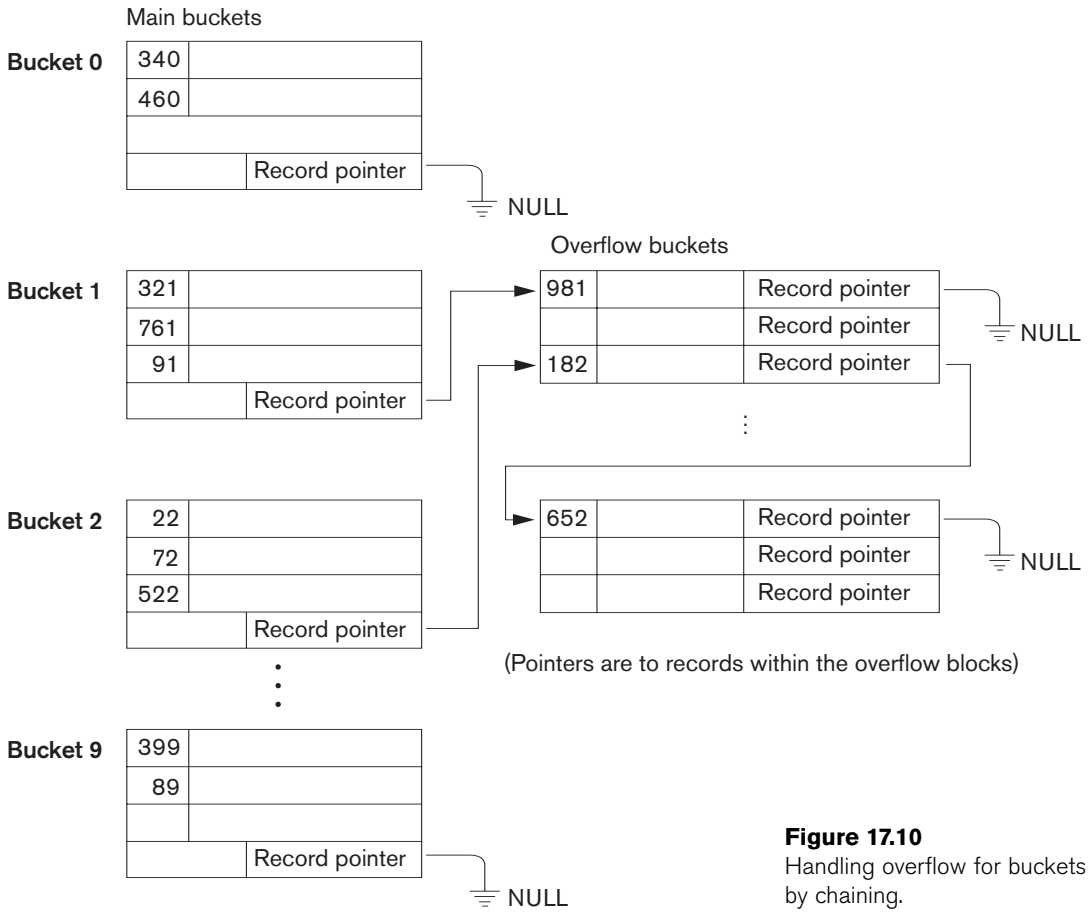


Figure 17.10
Handling overflow for buckets by chaining.

Modifying a specific record's field value depends on two factors: the search condition to locate that specific 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.

17.8.3 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 scheme—extendible hashing—stores an access structure in addition to the file, and

hence is somewhat similar to indexing (see Chapter 18). 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 values of the search field itself. The second technique, called linear hashing, does not require additional access structures. Another scheme, called **dynamic hashing**, uses an access structure based on binary tree data structures..

These hashing schemes take advantage of the fact that the result of applying a hashing function 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.

Extendible Hashing. In extendible hashing, a 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 **local depth d'**—stored with each bucket—specifies the number of bits on which the bucket contents are based. Figure 17.11 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 17.11. The records will be distributed between 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 17.11 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

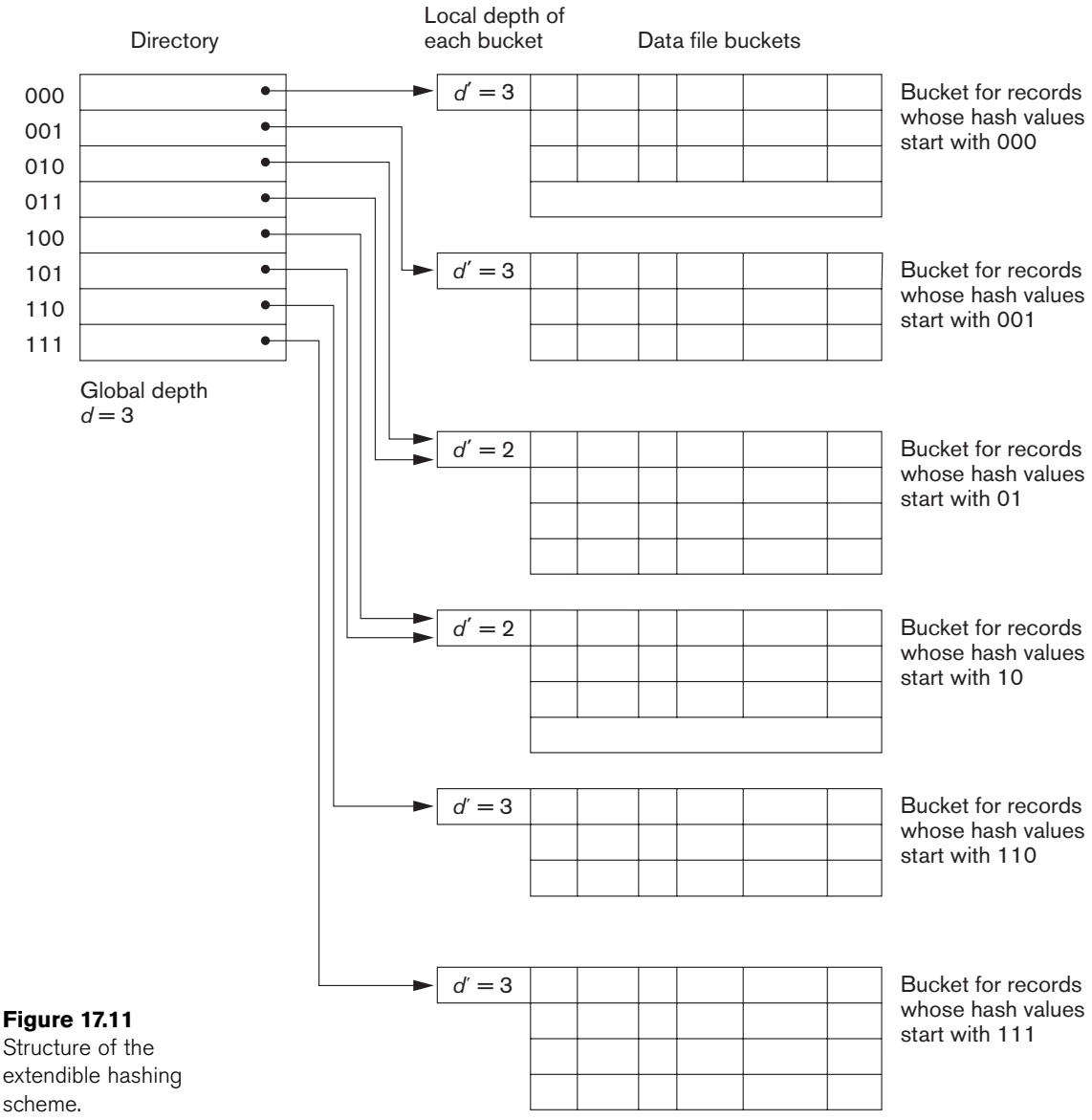


Figure 17.11
Structure of the extendible hashing scheme.

is also split into two locations, both of which have the same pointer value as did the original location.

The main advantage of extendible hashing that makes it attractive is that the performance of the file does not degrade as the file grows, as opposed to static external hashing where collisions increase and the corresponding chaining effectively

increases the average number of accesses per key. Additionally, no space is allocated in extendible hashing for future growth, but additional buckets can be allocated dynamically as needed. The space overhead for the directory table is negligible. The maximum directory size is 2^k, where k is the number of bits in the hash value. Another advantage is that splitting causes minor reorganization in most cases, since only the records in one bucket are redistributed to the two new buckets. The only time reorganization is more expensive is when the directory has to be doubled (or halved). A disadvantage is that the directory must be searched before accessing the buckets themselves, resulting in two block accesses instead of one in static hashing. This performance penalty is considered minor and thus the scheme is considered quite desirable for dynamic files.

Dynamic Hashing. A precursor to extendible hashing was dynamic hashing, in which the addresses of the buckets were either the n high-order bits or $n - 1$ high-order bits, depending on the total number of keys belonging to the respective bucket. The eventual storage of records in buckets for dynamic hashing is somewhat similar to extendible hashing. The major difference is in the organization of the directory. Whereas extendible hashing uses the notion of global depth (high-order d bits) for the flat directory and then combines adjacent collapsible buckets into a bucket of local depth $d - 1$, dynamic hashing maintains a tree-structured directory with two types of nodes:

- Internal nodes that have two pointers—the left pointer corresponding to the 0 bit (in the hashed address) and a right pointer corresponding to the 1 bit.
- Leaf nodes—these hold a pointer to the actual bucket with records.

An example of the dynamic hashing appears in Figure 17.12. Four buckets are shown ("000", "001", "110", and "111") with high-order 3-bit addresses (corresponding to the global depth of 3), and two buckets ("01" and "10") are shown with high-order 2-bit addresses (corresponding to the local depth of 2). The latter two are the result of collapsing the "010" and "011" into "01" and collapsing "100" and "101" into "10". Note that the directory nodes are used implicitly to determine the "global" and "local" depths of buckets in dynamic hashing. The search for a record given the hashed address involves traversing the directory tree, which leads to the bucket holding that record. It is left to the reader to develop algorithms for insertion, deletion, and searching of records for the dynamic hashing scheme.

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, ..., $M - 1$ and uses the mod hash function $h(K) = K \bmod M$; this hash function is called the **initial hash function** h_i. Overflow because of collisions is still needed and can be 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 between the two buckets based on a different hashing function $h_{i+1}(K) = K \bmod 2M$. A key property of the two hash

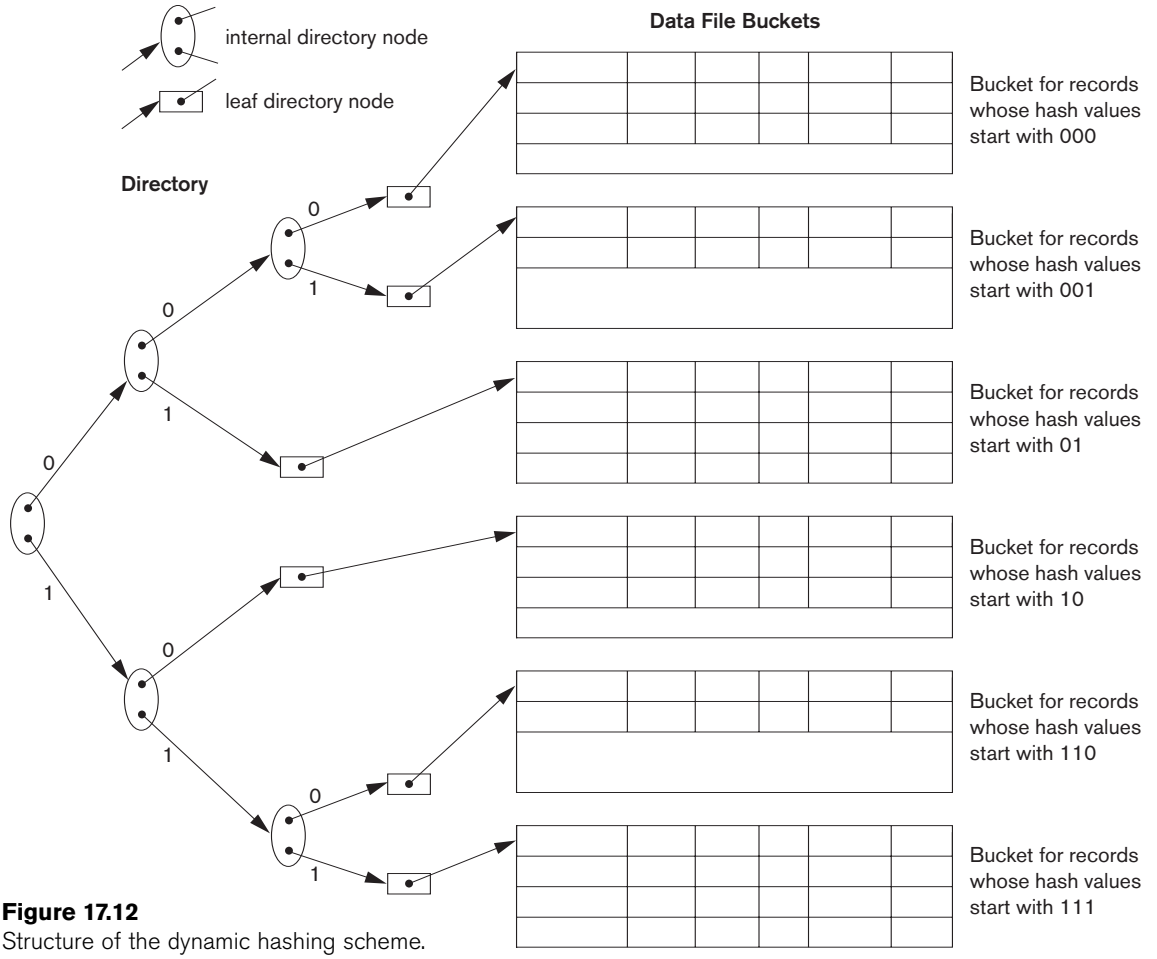


Figure 17.12
Structure of the dynamic hashing scheme.

functions h_i and h_{i+1} is that any records that hashed to bucket 0 based on h_i will hash to either bucket 0 or bucket M based on h_{i+1}; this is necessary for linear hashing to work.

As further collisions lead to overflow records, additional buckets are split in the *linear* order 1, 2, 3, ... If enough overflows occur, all the original file buckets 0, 1, ..., $M - 1$ will have been split, so the file now has $2M$ instead of M buckets, and all buckets use the hash function h_{i+1}. Hence, the records in overflow are eventually redistributed into regular buckets, using the function h_{i+1} via a *delayed split* of their buckets. There is no directory; only a value n—which is initially set to 0 and is incremented by 1 whenever a split occurs—is needed to determine which buckets have been split. To retrieve a record with hash key value K, first apply the function h_i to K; if $h_i(K) < n$, then apply the function h_{i+1} on K because the bucket is already split. Initially, $n = 0$, indicating that the function h_i applies to all buckets; n grows linearly as buckets are split.

When $n = M$ after being incremented, this signifies that all the original buckets have been split and the hash function h_{i+1} applies to all records in the file. At this point, n is reset to 0 (zero), and any new collisions that cause overflow lead to the use of a new hashing function $h_{i+2}(K) = K \bmod 4M$. In general, a sequence of hashing functions $h_{i+j}(K) = K \bmod (2^j M)$ is used, where $j = 0, 1, 2, \ldots$; a new hashing function h_{i+j+1} is needed whenever all the buckets $0, 1, \ldots, (2^j M) - 1$ have been split and n is reset to 0. The search for a record with hash key value K is given by Algorithm 17.3.

Splitting can be controlled by monitoring the file load factor instead of by splitting whenever an overflow occurs. In general, the **file load factor** l can be defined as $l = r/(bfr * N)$, 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. Buckets that have been split can also be recombined if the load factor of the file falls below a certain threshold. 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. The main advantages of linear hashing are that it maintains the load factor fairly constantly while the file grows and shrinks, and it does not require a directory.[10]

Algorithm 17.3. The Search Procedure for Linear Hashing

```
if n = 0
    then m ← h_j (K) (* m is the hash value of record with hash key K *)
    else begin
        m ← h_j (K);
        if m < n then m ← h_{j+1} (K)
        end;
```

search the bucket whose hash value is m (and its overflow, if any);

17.9 Other Primary File Organizations

17.9.1 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 7. Relationships among records in various files can be represented by **connecting fields**.[11] For example, a STUDENT record can have a connecting field Major_dept whose value gives the

[10]For details of insertion and deletion into Linear hashed files, refer to Litwin (1980) and Salzberg (1988).

[11]The concept of foreign keys in the relational data model (Chapter 3) and references among objects in object-oriented models (Chapter 11) are examples of connecting fields.

name of the DEPARTMENT in which the student is majoring. This Major_dept 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 object DBMSs, as well as legacy systems such as hierarchical and network DBMSs, often implement relationships among records as **physical relationships** realized by physical contiguity (or clustering) 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 types can be **physically clustered** on disk. If a particular relationship is expected to be used 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 frequent, it would be desirable to place each DEPARTMENT record and its cluster of STUDENT records contiguously on disk in a mixed file. The concept of **physical clustering** of object types is used in object DBMSs to store related objects together 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 data values in the record.

17.9.2 B-Trees and Other Data Structures as Primary Organization

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 a B-tree data structure as the primary file organization. We will describe B-trees in Section 18.3.1, 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. Recently, column-based storage of data has been proposed as a primary method for storage of relations in relational databases. We will briefly introduce it in Chapter 18 as a possible alternative storage scheme for relational databases.

17.10 Parallelizing Disk Access Using RAID Technology

With the exponential growth in the performance and capacity of semiconductor devices and memories, faster microprocessors with larger and larger primary memories are continually becoming available. To match this growth, it is natural to

expect that secondary storage technology must also take steps to keep up with processor technology in performance and reliability.

A major advance in secondary storage technology is represented by the development of **RAID**, which originally stood for **Redundant Arrays of Inexpensive Disks**. More recently, the *I* in RAID is said to stand for Independent. The RAID idea received a very positive industry endorsement and has been developed into an elaborate set of alternative RAID architectures (RAID levels 0 through 6). We highlight the main features of the technology in this section.

The main goal of RAID is to even out the widely different rates of performance improvement of disks against those in memory and microprocessors.[12] While RAM capacities have quadrupled every two to three years, disk *access times* are improving at less than 10 percent per year, and disk *transfer rates* are improving at roughly 20 percent per year. Disk *capacities* are indeed improving at more than 50 percent per year, but the speed and access time improvements are of a much smaller magnitude.

A second qualitative disparity exists between the ability of special microprocessors that cater to new applications involving video, audio, image, and spatial data processing (see Chapters 26 and 30 for details of these applications), with corresponding lack of fast access to large, shared data sets.

The natural solution is a large array of small independent disks acting as a single higher-performance logical disk. A concept called **data striping** is used, which utilizes *parallelism* to improve disk performance. Data striping distributes data transparently over multiple disks to make them appear as a single large, fast disk. Figure 17.13 shows a file distributed or *striped* over four disks. Striping improves overall I/O performance by allowing multiple I/Os to be serviced in parallel, thus providing high overall transfer rates. Data striping also accomplishes load balancing among disks. Moreover, by storing redundant information on disks using parity or some other error-correction code, reliability can be improved. In Sections 17.10.1 and

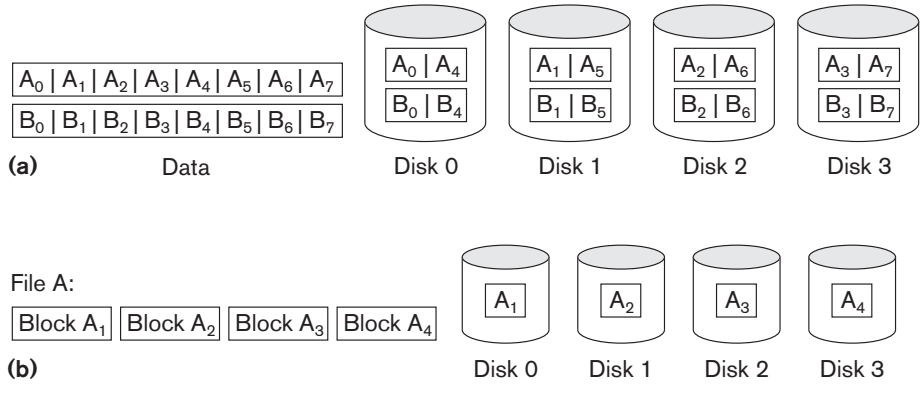


Figure 17.13
Striping of data across multiple disks. (a) Bit-level striping across four disks. (b) Block-level striping across four disks.

[12]This was predicted by Gordon Bell to be about 40 percent every year between 1974 and 1984 and is now supposed to exceed 50 percent per year.

17.10.2, we discuss how RAID achieves the two important objectives of improved reliability and higher performance. Section 17.10.3 discusses RAID organizations and levels.

17.10.1 Improving Reliability with RAID

For an array of n disks, the likelihood of failure is n times as much as that for one disk. Hence, if the MTBF (Mean Time Between Failures) of a disk drive is assumed to be 200,000 hours or about 22.8 years (for the disk drive in Table 17.1 called Cheetah NS, it is 1.4 million hours), the MTBF for a bank of 100 disk drives becomes only 2,000 hours or 83.3 days (for 1,000 Cheetah NS disks it would be 1,400 hours or 58.33 days). Keeping a single copy of data in such an array of disks will cause a significant loss of reliability. An obvious solution is to employ redundancy of data so that disk failures can be tolerated. The disadvantages are many: additional I/O operations for write, extra computation to maintain redundancy and to do recovery from errors, and additional disk capacity to store redundant information.

One technique for introducing redundancy is called **mirroring** or **shadowing**. Data is written redundantly to two identical physical disks that are treated as one logical disk. When data is read, it can be retrieved from the disk with shorter queuing, seek, and rotational delays. If a disk fails, the other disk is used until the first is repaired. Suppose the mean time to repair is 24 hours, then the mean time to data loss of a mirrored disk system using 100 disks with MTBF of 200,000 hours each is $(200{,}000)^2/(2 * 24) = 8.33 * 10^8$ hours, which is 95,028 years.[13] Disk mirroring also doubles the rate at which read requests are handled, since a read can go to either disk. The transfer rate of each read, however, remains the same as that for a single disk.

Another solution to the problem of reliability is to store extra information that is not normally needed but that can be used to reconstruct the lost information in case of disk failure. The incorporation of redundancy must consider two problems: selecting a technique for computing the redundant information, and selecting a method of distributing the redundant information across the disk array. The first problem is addressed by using error-correcting codes involving parity bits, or specialized codes such as Hamming codes. Under the parity scheme, a redundant disk may be considered as having the sum of all the data in the other disks. When a disk fails, the missing information can be constructed by a process similar to subtraction.

For the second problem, the two major approaches are either to store the redundant information on a small number of disks or to distribute it uniformly across all disks. The latter results in better load balancing. The different levels of RAID choose a combination of these options to implement redundancy and improve reliability.

17.10.2 Improving Performance with RAID

The disk arrays employ the technique of data striping to achieve higher transfer rates. Note that data can be read or written only one block at a time, so a typical transfer contains 512 to 8192 bytes. Disk striping may be applied at a finer granularity by

[13]The formulas for MTBF calculations appear in Chen et al. (1994).

breaking up a byte of data into bits and spreading the bits to different disks. Thus, **bit-level data striping** consists of splitting a byte of data and writing bit *j* to the *j*th disk. With 8-bit bytes, eight physical disks may be considered as one logical disk with an eightfold increase in the data transfer rate. Each disk participates in each I/O request and the total amount of data read per request is eight times as much. Bit-level striping can be generalized to a number of disks that is either a multiple or a factor of eight. Thus, in a four-disk array, bit *n* goes to the disk which is (*n* mod 4). Figure 17.13(a) shows bit-level striping of data.

The granularity of data interleaving can be higher than a bit; for example, blocks of a file can be striped across disks, giving rise to **block-level striping**. Figure 17.13(b) shows block-level data striping assuming the data file contains four blocks. With block-level striping, multiple independent requests that access single blocks (small requests) can be serviced in parallel by separate disks, thus decreasing the queuing time of I/O requests. Requests that access multiple blocks (large requests) can be parallelized, thus reducing their response time. In general, the more the number of disks in an array, the larger the potential performance benefit. However, assuming independent failures, the disk array of 100 disks collectively has 1/100th the reliability of a single disk. Thus, redundancy via error-correcting codes and disk mirroring is necessary to provide reliability along with high performance.

17.10.3 RAID Organizations and Levels

Different RAID organizations were defined based on different combinations of the two factors of granularity of data interleaving (striping) and pattern used to compute redundant information. In the initial proposal, levels 1 through 5 of RAID were proposed, and two additional levels—0 and 6—were added later.

RAID level 0 uses data striping, has no redundant data, and hence has the best write performance since updates do not have to be duplicated. It splits data evenly across two or more disks. However, its read performance is not as good as RAID level 1, which uses mirrored disks. In the latter, performance improvement is possible by scheduling a read request to the disk with shortest expected seek and rotational delay. RAID level 2 uses memory-style redundancy by using Hamming codes, which contain parity bits for distinct overlapping subsets of components. Thus, in one particular version of this level, three redundant disks suffice for four original disks, whereas with mirroring—as in level 1—four would be required. Level 2 includes both error detection and correction, although detection is generally not required because broken disks identify themselves.

RAID level 3 uses a single parity disk relying on the disk controller to figure out which disk has failed. Levels 4 and 5 use block-level data striping, with level 5 distributing data and parity information across all disks. Figure 17.14(b) shows an illustration of RAID level 5, where parity is shown with subscript p. If one disk fails, the missing data is calculated based on the parity available from the remaining disks. Finally, RAID level 6 applies the so-called $P + Q$ redundancy scheme using Reed-Soloman codes to protect against up to two disk failures by using just two redundant disks.

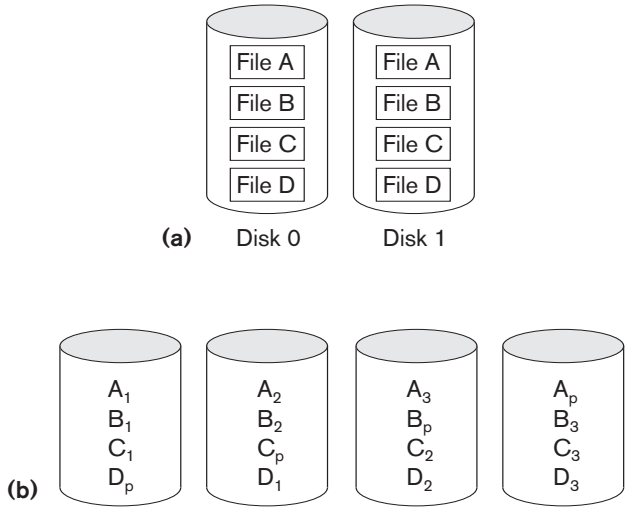


Figure 17.14
Some popular levels of RAID. (a) RAID level 1: Mirroring of data on two disks. (b) RAID level 5: Striping of data with distributed parity across four disks.

Rebuilding in case of disk failure is easiest for RAID level 1. Other levels require the reconstruction of a failed disk by reading multiple disks. Level 1 is used for critical applications such as storing logs of transactions. Levels 3 and 5 are preferred for large volume storage, with level 3 providing higher transfer rates. Most popular use of RAID technology currently uses level 0 (with striping), level 1 (with mirroring), and level 5 with an extra drive for parity. A combination of multiple RAID levels are also used – for example, 0+1 combines striping and mirroring using a minimum of four disks. Other nonstandard RAID levels include: RAID 1.5, RAID 7, RAID-DP, RAID S or Parity RAID, Matrix RAID, RAID-K, RAID-Z, RAIDn, Linux MD RAID 10, IBM ServeRAID 1E, and unRAID. A discussion of these nonstandard levels is beyond the scope of this book. Designers of a RAID setup for a given application mix have to confront many design decisions such as the level of RAID, the number of disks, the choice of parity schemes, and grouping of disks for block-level striping. Detailed performance studies on small reads and writes (referring to I/O requests for one striping unit) and large reads and writes (referring to I/O requests for one stripe unit from each disk in an error-correction group) have been performed.

17.11 New Storage Systems

In this section, we describe three recent developments in storage systems that are becoming an integral part of most enterprise's information system architectures.

17.11.1 Storage Area Networks

With the rapid growth of electronic commerce, Enterprise Resource Planning (ERP) systems that integrate application data across organizations, and data warehouses that keep historical aggregate information (see Chapter 29), the demand for storage has gone up substantially. For today's Internet-driven organizations, it has

become necessary to move from a static fixed data center-oriented operation to a more flexible and dynamic infrastructure for their information processing requirements. The total cost of managing all data is growing so rapidly that in many instances the cost of managing server-attached storage exceeds the cost of the server itself. Furthermore, the procurement cost of storage is only a small fraction—typically, only 10 to 15 percent of the overall cost of storage management. Many users of RAID systems cannot use the capacity effectively because it has to be attached in a fixed manner to one or more servers. Therefore, most large organizations have moved to a concept called **storage area networks (SANs)**. In a SAN, online storage peripherals are configured as nodes on a high-speed network and can be attached and detached from servers in a very flexible manner. Several companies have emerged as SAN providers and supply their own proprietary topologies. They allow storage systems to be placed at longer distances from the servers and provide different performance and connectivity options. Existing storage management applications can be ported into SAN configurations using Fiber Channel networks that encapsulate the legacy SCSI protocol. As a result, the SAN-attached devices appear as SCSI devices.

Current architectural alternatives for SAN include the following: point-to-point connections between servers and storage systems via fiber channel; use of a fiber channel switch to connect multiple RAID systems, tape libraries, and so on to servers; and the use of fiber channel hubs and switches to connect servers and storage systems in different configurations. Organizations can slowly move up from simpler topologies to more complex ones by adding servers and storage devices as needed. We do not provide further details here because they vary among SAN vendors. The main advantages claimed include:

- Flexible many-to-many connectivity among servers and storage devices using fiber channel hubs and switches
- Up to 10 km separation between a server and a storage system using appropriate fiber optic cables
- Better isolation capabilities allowing nondisruptive addition of new peripherals and servers

SANs are growing very rapidly, but are still faced with many problems, such as combining storage options from multiple vendors and dealing with evolving standards of storage management software and hardware. Most major companies are evaluating SANs as a viable option for database storage.

17.11.2 Network-Attached Storage

With the phenomenal growth in digital data, particularly generated from multimedia and other enterprise applications, the need for high-performance storage solutions at low cost has become extremely important. **Network-attached storage** (NAS) devices are among the storage devices being used for this purpose. These devices are, in fact, servers that do not provide any of the common server services, but simply allow the addition of storage for file sharing. NAS devices allow vast

amounts of hard-disk storage space to be added to a network and can make that space available to multiple servers without shutting them down for maintenance and upgrades. NAS devices can reside anywhere on a local area network (LAN) and may be combined in different configurations. A single hardware device, often called the **NAS box** or **NAS head**, acts as the interface between the NAS system and network clients. These NAS devices require no monitor, keyboard, or mouse. One or more disk or tape drives can be attached to many NAS systems to increase total capacity. Clients connect to the NAS head rather than to the individual storage devices. An NAS can store any data that appears in the form of files, such as e-mail boxes, Web content, remote system backups, and so on. In that sense, NAS devices are being deployed as a replacement for traditional file servers.

NAS systems strive for reliable operation and easy administration. They include built-in features such as secure authentication, or the automatic sending of e-mail alerts in case of error on the device. The NAS devices (or *appliances*, as some vendors refer to them) are being offered with a high degree of scalability, reliability, flexibility, and performance. Such devices typically support RAID levels 0, 1, and 5. Traditional storage area networks (SANs) differ from NAS in several ways. Specifically, SANs often utilize Fiber Channel rather than Ethernet, and a SAN often incorporates multiple network devices or *endpoints* on a self-contained or *private* LAN, whereas NAS relies on individual devices connected directly to the existing public LAN. Whereas Windows, UNIX, and NetWare file servers each demand specific protocol support on the client side, NAS systems claim greater operating system independence of clients.

17.11.3 iSCSI Storage Systems

A new protocol called **iSCSI** (Internet SCSI) has been proposed recently. It allows clients (called *initiators*) to send SCSI commands to SCSI storage devices on remote channels. The main advantage of iSCSI is that it does not require the special cabling needed by Fiber Channel and it can run over longer distances using existing network infrastructure. By carrying SCSI commands over IP networks, iSCSI facilitates data transfers over intranets and manages storage over long distances. It can transfer data over local area networks (LANs), wide area networks (WANs), or the Internet.

iSCSI works as follows. When a DBMS needs to access data, the operating system generates the appropriate SCSI commands and data request, which then go through encapsulation and, if necessary, encryption procedures. A packet header is added before the resulting IP packets are transmitted over an Ethernet connection. When a packet is received, it is decrypted (if it was encrypted before transmission) and disassembled, separating the SCSI commands and request. The SCSI commands go via the SCSI controller to the SCSI storage device. Because iSCSI is bidirectional, the protocol can also be used to return data in response to the original request. Cisco and IBM have marketed switches and routers based on this technology.

iSCSI storage has mainly impacted small- and medium-sized businesses because of its combination of simplicity, low cost, and the functionality of iSCSI devices. It allows them not to learn the ins and outs of Fiber Channel (FC) technology and

instead benefit from their familiarity with the IP protocol and Ethernet hardware. iSCSI implementations in the data centers of very large enterprise businesses are slow in development due to their prior investment in Fiber Channel-based SANs.

iSCSI is one of two main approaches to storage data transmission over IP networks. The other method, **Fiber Channel over IP (FCIP)**, translates Fiber Channel control codes and data into IP packets for transmission between geographically distant Fiber Channel storage area networks. This protocol, known also as *Fiber Channel tunneling* or *storage tunneling*, can only be used in conjunction with Fiber Channel technology, whereas iSCSI can run over existing Ethernet networks.

The latest idea to enter the enterprise IP storage race is **Fiber Channel over Ethernet (FCoE)**, which can be thought of as iSCSI without the IP. It uses many elements of SCSI and FC (just like iSCSI), but it does not include TCP/IP components. This promises excellent performance, especially on 10 Gigabit Ethernet (10GbE), and is relatively easy for vendors to add to their products.

17.12 Summary

We began this chapter by discussing the characteristics of memory hierarchies and then concentrated on secondary storage devices. In particular, we focused on magnetic disks because they are used most often to store online 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. To reduce the average block access time, double buffering can be used when accessing consecutive disk blocks. (Other disk parameters are discussed in Appendix B.) We presented different ways of storing file records on disk. File records are grouped into disk blocks and can be fixed length or variable length, spanned or unspanned, and of the same record type or mixed types. 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.

Then we 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 key. 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 ordered access of the records on any field. We discussed three hashing techniques for files that grow and shrink in the number of records dynamically: extendible, dynamic, and linear hashing. The first two use the higher-order bits of the hash address to organize a directory. Linear hashing is geared to keep the load factor of the file within a given range and adds new buckets linearly.

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. We reviewed the recent advances in disk technology represented by RAID (Redundant Arrays of Inexpensive (or Independent) Disks), which has become a standard technique in large enterprises to provide better reliability and fault tolerance features in storage. Finally, we reviewed three currently popular options in enterprise storage systems: storage area networks (SANs), network-attached storage (NAS), and iSCSI storage systems.

Review Questions

17.1. What is the difference between primary and secondary storage?

17.2. Why are disks, not tapes, used to store online database files?

17.3. Define the following terms: *disk, disk pack, track, block, cylinder, sector, interblock gap, read/write head.*

17.4. Discuss the process of disk initialization.

17.5. Discuss the mechanism used to read data from or write data to the disk.

17.6. What are the components of a disk block address?

17.7. Why is accessing a disk block expensive? Discuss the time components involved in accessing a disk block.

17.8. How does double buffering improve block access time?

17.9. What are the reasons for having variable-length records? What types of separator characters are needed for each?

17.10. Discuss the techniques for allocating file blocks on disk.

17.11. What is the difference between a file organization and an access method?

17.12. What is the difference between static and dynamic files?

17.13. What are the typical record-at-a-time operations for accessing a file? Which of these depend on the current file record?

17.14. Discuss the techniques for record deletion.

17.15. Discuss the advantages and disadvantages of using (a) an unordered file, (b) an ordered file, and (c) a static hash file with buckets and chaining. Which operations can be performed efficiently on each of these organizations, and which operations are expensive?

17.16. Discuss the techniques for allowing a hash file to expand and shrink dynamically. What are the advantages and disadvantages of each?

17.17. What is the difference between the directories of extendible and dynamic hashing?

17.18. What are mixed files used for? What are other types of primary file organizations?

17.19. Describe the mismatch between processor and disk technologies.

17.20. What are the main goals of the RAID technology? How does it achieve them?

17.21. How does disk mirroring help improve reliability? Give a quantitative example.

17.22. What characterizes the levels in RAID organization?

17.23. What are the highlights of the popular RAID levels 0, 1, and 5?

17.24. What are storage area networks? What flexibility and advantages do they offer?

17.25. Describe the main features of network-attached storage as an enterprise storage solution.

17.26. How have new iSCSI systems improved the applicability of storage area networks?

Exercises

17.27. 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 (*tr*) 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.

17.28. 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 (10 bytes), Birth_date (8 bytes), Sex (1 byte), Major_dept_code (4 bytes), Minor_dept_code (4 bytes), Class_code (4 bytes, integer), and Degree_program (3 bytes). An additional byte is used as a deletion marker. The file is stored on the disk whose parameters are given in Exercise 17.27.

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; by doing a binary search, calculate the time it takes to search for a record given its Ssn value.

17.29. Suppose that only 80 percent of the STUDENT records from Exercise 17.28 have a value for Phone, 85 percent for Major_dept_code, 15 percent for Minor_dept_code, and 90 percent for Degree_program; and suppose that we use a variable-length record file. Each record has a 1-byte *field type* for each field 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.

17.30. 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; Last_name, 20 bytes; First_name, 20 bytes; Middle_init, 1 byte; Birth_date, 10 bytes; Address, 35 bytes; Phone, 12 bytes; Supervisor_ssn, 9 bytes; Department, 4 bytes; Job_code, 4 bytes; deletion marker, 1 byte. The EMPLOYEE file has r = 30,000 records, fixed-length format, and unspanned blocking. Write 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 in msec the average *time* needed 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 in msec the average *time* needed 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.

17.31. A PARTS file with Part# as the hash key includes records with the following Part# values: 2369, 3760, 4692, 4871, 5659, 1821, 1074, 7115, 1620, 2428, 3943, 4750, 6975, 4981, and 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#.

17.32. Load the records of Exercise 17.31 into expandable hash files based on extendible hashing. Show the structure of the directory at each step, and the global and local depths. Use the hash function $h(K) = K \bmod 128$.

17.33. Load the records of Exercise 17.31 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.

17.34. Compare the file commands listed in Section 17.5 to those available on a file access method you are familiar with.

17.35. 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.

17.36. 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.

17.37. Can you think of techniques other than an unordered overflow file that can be used to make insertions in an ordered file more efficient?

17.38. 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.

17.39. Can you think of techniques other than chaining to handle bucket overflow in external hashing?

17.40. Write pseudocode for the insertion algorithms for linear hashing and for extendible hashing.

17.41. Write program code to access individual fields of records under each of the following circumstances. For each case, state the assumptions you make concerning pointers, separator characters, and so on. 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

17.42. Suppose that a file initially contains $r = 120{,}000$ records of $R = 200$ bytes each in an unsorted (heap) file. The block size $B = 2400$ bytes, the average seek time $s = 16$ ms, the average rotational latency $rd = 8.3$ ms, and the block transfer time $btt = 0.8$ ms. Assume that 1 record is deleted for every 2 records added until the total number of active records is 240,000.

a. How many block transfers are needed to reorganize the file?
b. How long does it take to find a record right before reorganization?
c. How long does it take to find a record right after reorganization?

17.43. Suppose we have a sequential (ordered) file of 100,000 records where each record is 240 bytes. Assume that $B = 2400$ bytes, $s = 16$ ms, $rd = 8.3$ ms, and $btt = 0.8$ ms. Suppose we want to make X independent random record reads from the file. We could make X random block reads or we could perform one exhaustive read of the entire file looking for those X records. The question is to decide when it would be more efficient to perform one exhaustive read of the entire file than to perform X individual random reads. That is, what is the value for X when an exhaustive read of the file is more efficient than random X reads? Develop this as a function of X.

17.44. Suppose that a static hash file initially has 600 buckets in the primary area and that records are inserted that create an overflow area of 600 buckets. If we reorganize the hash file, we can assume that most of the overflow is eliminated. If the cost of reorganizing the file is the cost of the bucket transfers (reading and writing all of the buckets) and the only periodic file operation is the fetch operation, then how many times would we have to perform a fetch (successfully) to make the reorganization cost effective? That is, the reorganization cost and subsequent search cost are less than the search cost before reorganization. Support your answer. Assume $s = 16$ ms, $rd = 8.3$ ms, and $btt = 1$ ms.

17.45. Suppose we want to create a linear hash file with a file load factor of 0.7 and a blocking factor of 20 records per bucket, which is to contain 112,000 records initially.

a. How many buckets should we allocate in the primary area?

b. What should be the number of bits used for bucket addresses?

Selected Bibliography

Wiederhold (1987) has a detailed discussion and analysis of secondary storage devices and file organizations as a part of database design. Optical disks are described in Berg and Roth (1989) and analyzed in Ford and Christodoulakis (1991). Flash memory is discussed by Dipert and Levy (1993). Ruemmler and Wilkes (1994) present a survey of the magnetic-disk technology. Most textbooks on databases include discussions of the material presented here. Most data structures textbooks, including Knuth (1998), 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. Textbooks on file structures include Claybrook (1992), Smith and Barnes (1987), and Salzberg (1988); they discuss additional file organizations including tree-structured files, and have detailed algorithms for operations on files. Salzberg et al. (1990) describe a distributed external sorting algorithm. File organizations with a high degree of fault tolerance are described by Bitton and Gray (1988) and by Gray et al. (1990). Disk striping was proposed in Salem and Garcia Molina (1986). The first paper on redundant arrays of inexpensive disks (RAID) is by Patterson et al. (1988). Chen and Patterson (1990) and the excellent survey of RAID by Chen et al. (1994) are additional references. Grochowski and Hoyt (1996) discuss future trends in disk drives. Various formulas for the RAID architecture appear in Chen et al. (1994).

Morris (1968) is an early paper on hashing. Extendible hashing is described in Fagin et al. (1979). Linear hashing is described by Litwin (1980). Algorithms for insertion and deletion for linear hashing are discussed with illustrations in Salzberg (1988). Dynamic hashing, which we briefly introduced, was proposed by Larson (1978). There are many proposed variations for extendible and linear hashing; for examples, see Cesarini and Soda (1991), Du and Tong (1991), and Hachem and Berra (1992).

Details of disk storage devices can be found at manufacturer sites (for example, http://www.seagate.com, http://www.ibm.com, http://www.emc.com, http://www.hp.com, http://www.storagetek.com,. IBM has a storage technology research center at IBM Almaden (http://www.almaden.ibm.com/).

chapter 18

Indexing Structures for Files

In this chapter we assume that a file already exists with some primary organization such as the unordered, ordered, or hashed organizations that were described in Chapter 17. We will describe additional auxiliary **access structures** called **indexes**, which are used to speed up the retrieval of records in response to certain search conditions. The index structures are additional files on disk that provide **secondary access paths**, which provide alternative ways to access the records without affecting the physical placement of records in the primary data file on disk. They enable efficient access to records based on the **indexing fields** that are used to construct the index. Basically, *any field* of the file can be used to create an index, and *multiple indexes* on different fields—as well as indexes on *multiple fields*—can be constructed on the same file. A variety of indexes are possible; each of them uses a particular data structure to speed up the search. To find a record or records in the data file based on a search condition on an indexing field, the index is searched, which leads to pointers to one or more disk blocks in the data file where the required records are located. The most prevalent types of indexes are based on ordered files (single-level indexes) and tree data structures (multilevel indexes, B^+-trees). Indexes can also be constructed based on hashing or other search data structures. We also discuss indexes that are vectors of bits called *bitmap indexes*.

We describe different types of single-level ordered indexes—primary, secondary, and clustering—in Section 18.1. By viewing a single-level index as an ordered file, one can develop additional indexes for it, giving rise to the concept of multilevel indexes. A popular indexing scheme called **ISAM (Indexed Sequential Access Method)** is based on this idea. We discuss multilevel tree-structured indexes in Section 18.2. In Section 18.3 we describe B-trees and B^+-trees, which are data structures that are commonly used in DBMSs to implement dynamically changing multilevel indexes. B^+-trees have become a commonly accepted default structure for

generating indexes on demand in most relational DBMSs. Section 18.4 is devoted to alternative ways to access data based on a combination of multiple keys. In Section 18.5 we discuss hash indexes and introduce the concept of logical indexes, which give an additional level of indirection from physical indexes, allowing for the physical index to be flexible and extensible in its organization. In Section 18.6 we discuss multikey indexing and bitmap indexes used for searching on one or more keys. Section 18.7 summarizes the chapter.

18.1 Types of Single-Level Ordered Indexes

The idea behind an ordered index is similar to that behind the index used in a textbook, which lists important terms at the end of the book in alphabetical order along with a list of page numbers where the term appears in the book. We can search the book index for a certain term in the textbook to find a list of *addresses*—page numbers in this case—and use these addresses to locate the specified pages first and then *search* for the term on each specified page. The alternative, if no other guidance is given, would be to sift slowly through the whole textbook word by word to find the term we are interested in; this corresponds to doing a *linear search*, which scans the whole file. Of course, most books do have additional information, such as chapter and section titles, which help us find a term without having to search through the whole book. However, the index is the only exact indication of the pages where each term occurs in the book.

For a file with a given record structure consisting of several fields (or attributes), an index access structure is usually defined on a single field of a file, called an **indexing field** (or **indexing attribute**).[1] The index typically stores each value of the index field along with a list of pointers to all disk blocks that contain records with that field value. The values in the index are *ordered* so that we can do a *binary search* on the index. If both the data file and the index file are ordered, and since the index file is typically much smaller than the data file, searching the index using a binary search is a better option. Tree-structured multilevel indexes (see Section 18.2) implement an extension of the binary search idea that reduces the search space by 2-way partitioning at each search step, thereby creating a more efficient approach that divides the search space in the file *n*-ways at each stage.

There are several types of ordered indexes. A **primary index** is specified on the *ordering key field* of an **ordered file** of records. Recall from Section 17.7 that an ordering key field is used to *physically order* the file records on disk, and every record has a *unique value* for that field. If the ordering field is not a key field—that is, if numerous records in the file can have the same value for the ordering field—another type of index, called a **clustering index**, can be used. The data file is called a **clustered file** in this latter case. Notice that a file can have at most one physical ordering field, so it can have at most one primary index or one clustering index, *but not both*. A third type of index, called a **secondary index**, can be specified on any

[1]We use the terms *field* and *attribute* interchangeably in this chapter.

nonordering field of a file. A data file can have several secondary indexes in addition to its primary access method. We discuss these types of single-level indexes in the next three subsections.

18.1.1 Primary Indexes

A **primary index** is an ordered file whose records are of fixed length with two fields, and it acts like an access structure to efficiently search for and access the data records in a data file. The first field is of the same data type as the ordering key field—called the **primary key**—of the data file, and the second field is a pointer to a disk block (a block address). There is one **index entry** (or **index record**) in the index file for each *block* in the data file. Each index entry has the value of the primary key field for the *first* record in a block and a pointer to that block as its two field values. We will refer to the two field values of index entry i as $<K(i), P(i)>$.

To create a primary index on the ordered file shown in Figure 17.7, we use the Name field as primary key, because that is the ordering key field of the file (assuming that each value of Name is unique). Each entry in the index has a Name value and a pointer. The first three index entries are as follows:

$<K(1) = (\text{Aaron, Ed}), P(1) = \text{address of block 1}>$

$<K(2) = (\text{Adams, John}), P(2) = \text{address of block 2}>$

$<K(3) = (\text{Alexander, Ed}), P(3) = \text{address of block 3}>$

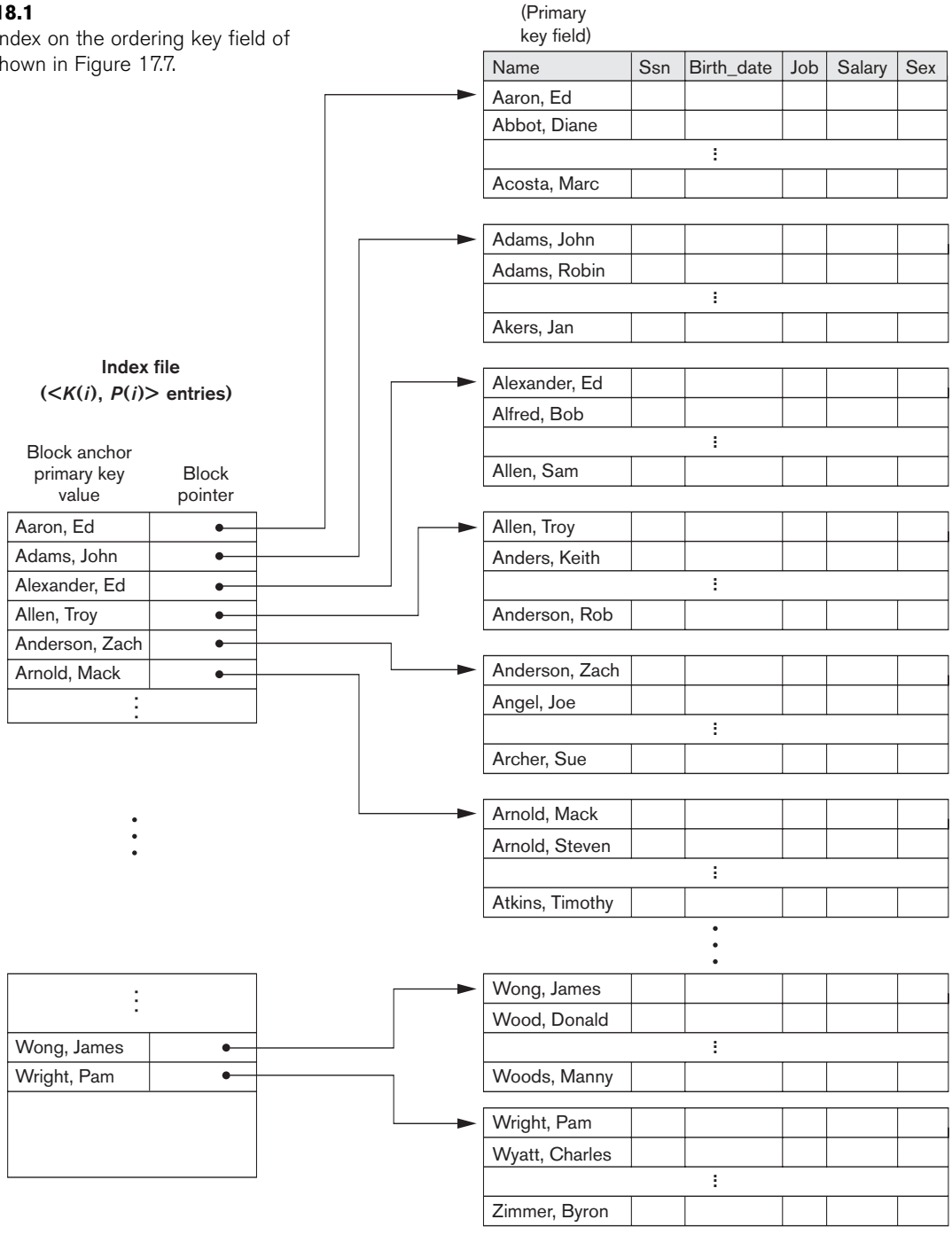
Figure 18.1 illustrates this primary index. The total number of entries in the index is the same as the *number of disk blocks* in the ordered data file. The first record in each block of the data file is called the **anchor record** of the block, or simply the **block anchor**.[2]

Indexes can also be characterized as dense or sparse. A **dense index** has an index entry for *every search key value* (and hence every record) in the data file. A **sparse** (or **nondense**) **index**, on the other hand, has index entries for only some of the search values. A sparse index has fewer entries than the number of records in the file. Thus, a primary index is a nondense (sparse) index, since it includes an entry for each disk block of the data file and the keys of its anchor record rather than for every search value (or every record).

The index file for a primary index occupies a much smaller space than does the data file, for two reasons. First, there are *fewer index entries* than there are records in the data file. Second, each index entry is typically *smaller in size* than a data record because it has only two fields; consequently, more index entries than data records can fit in one block. Therefore, a binary search on the index file requires fewer block accesses than a binary search on the data file. Referring to Table 17.2, note that the binary search for an ordered data file required $\log_2 b$ block accesses. But if the primary index file contains only b_i blocks, then to locate a record with a search key

[2]We can use a scheme similar to the one described here, with the last record in each block (rather than the first) as the block anchor. This slightly improves the efficiency of the search algorithm.

Figure 18.1
Primary index on the ordering key field of the file shown in Figure 17.7.



value requires a binary search of that index and access to the block containing that record: a total of $\log_2 b_i + 1$ accesses.

A record whose primary key value is K lies in the block whose address is $P(i)$, where $K(i) \leq K < K(i+1)$. The ith block in the data file contains all such records because of the physical ordering of the file records on the primary key field. To retrieve a record, given the value K of its primary key field, we do a binary search on the index file to find the appropriate index entry i, and then retrieve the data file block whose address is $P(i)$.[3] Example 1 illustrates the saving in block accesses that is attainable when a primary index is used to search for a record.

Example 1. Suppose that we have an ordered file with $r = 30{,}000$ records stored on a disk with block size $B = 1024$ bytes. File records are of fixed size and are unspanned, with record length $R = 100$ bytes. The blocking factor for the file would be $bfr = \lfloor (B/R) \rfloor = \lfloor (1024/100) \rfloor = 10$ records per block. The number of blocks needed for the file is $b = \lceil (r/bfr) \rceil = \lceil (30000/10) \rceil = 3000$ blocks. A binary search on the data file would need approximately $\lceil \log_2 b \rceil = \lceil (\log_2 3000) \rceil = 12$ block accesses.

Now suppose that the ordering key field of the file is $V = 9$ bytes long, a block pointer is $P = 6$ bytes long, and we have constructed a primary index for the file. The size of each index entry is $R_i = (9 + 6) = 15$ bytes, so the blocking factor for the index is $bfr_i = \lfloor (B/R_i) \rfloor = \lfloor (1024/15) \rfloor = 68$ entries per block. The total number of index entries r_i is equal to the number of blocks in the data file, which is 3000. The number of index blocks is hence $b_i = \lceil (r_i/bfr_i) \rceil = \lceil (3000/68) \rceil = 45$ blocks. To perform a binary search on the index file would need $\lceil (\log_2 b_i) \rceil = \lceil (\log_2 45) \rceil = 6$ block accesses. To search for a record using the index, we need one additional block access to the data file for a total of $6 + 1 = 7$ block accesses—an improvement over binary search on the data file, which required 12 disk block accesses.

A major problem with a primary index—as with any ordered file—is insertion and deletion of records. With a primary index, the problem is compounded because if we attempt to insert a record in its correct position in the data file, we must not only move records to make space for the new record but also change some index entries, since moving records will change the *anchor records* of some blocks. Using an unordered overflow file, as discussed in Section 17.7, can reduce this problem. Another possibility is to use a linked list of overflow records for each block in the data file. This is similar to the method of dealing with overflow records described with hashing in Section 17.8.2. Records within each block and its overflow linked list can be sorted to improve retrieval time. Record deletion is handled using deletion markers.

18.1.2 Clustering Indexes

If file records are physically ordered on a nonkey field—which *does not* have a distinct value for each record—that field is called the **clustering field** and the data file

[3]Notice that the above formula would not be correct if the data file were ordered on a *nonkey field*; in that case the same index value in the block anchor could be repeated in the last records of the previous block.

is called a **clustered file.** We can create a different type of index, called a **clustering index**, to speed up retrieval of all the records that have the same value for the clustering field. This differs from a primary index, which requires that the ordering field of the data file have a *distinct value* for each record.

A clustering index is also an ordered file with two fields; the first field is of the same type as the clustering field of the data file, and the second field is a disk block pointer. There is one entry in the clustering index for each *distinct value* of the clustering field, and it contains the value and a pointer to the *first block* in the data file that has a record with that value for its clustering field. Figure 18.2 shows an example. Notice that record insertion and deletion still cause problems because the data records are physically ordered. To alleviate the problem of insertion, it is common to reserve a whole block (or a cluster of contiguous blocks) for *each value* of the clustering field; all records with that value are placed in the block (or block cluster). This makes insertion and deletion relatively straightforward. Figure 18.3 shows this scheme.

A clustering index is another example of a *nondense* index because it has an entry for every *distinct value* of the indexing field, which is a nonkey by definition and hence has duplicate values rather than a unique value for every record in the file. There is some similarity between Figures 18.1, 18.2, and 18.3 and Figures 17.11 and 17.12. An index is somewhat similar to dynamic hashing (described in Section 17.8.3) and to the directory structures used for extendible hashing. Both are searched to find a pointer to the data block containing the desired record. A main difference is that an index search uses the values of the search field itself, whereas a hash directory search uses the binary hash value that is calculated by applying the hash function to the search field.

18.1.3 Secondary Indexes

A **secondary index** provides a secondary means of accessing a data file for which some primary access already exists. The data file records could be ordered, unordered, or hashed. The secondary index may be created on a field that is a candidate key and has a unique value in every record, or on a nonkey field with duplicate values. The index is again an ordered file with two fields. The first field is of the same data type as some *nonordering field* of the data file that is an **indexing field**. The second field is either a *block* pointer or a *record* pointer. *Many* secondary indexes (and hence, indexing fields) can be created for the same file—each represents an additional means of accessing that file based on some specific field.

First we consider a secondary index access structure on a key (unique) field that has a *distinct value* for every record. Such a field is sometimes called a **secondary** key; in the relational model, this would correspond to any UNIQUE key attribute or to the primary key attribute of a table. In this case there is one index entry for *each record* in the data file, which contains the value of the field for the record and a pointer either to the block in which the record is stored or to the record itself. Hence, such an index is **dense.**

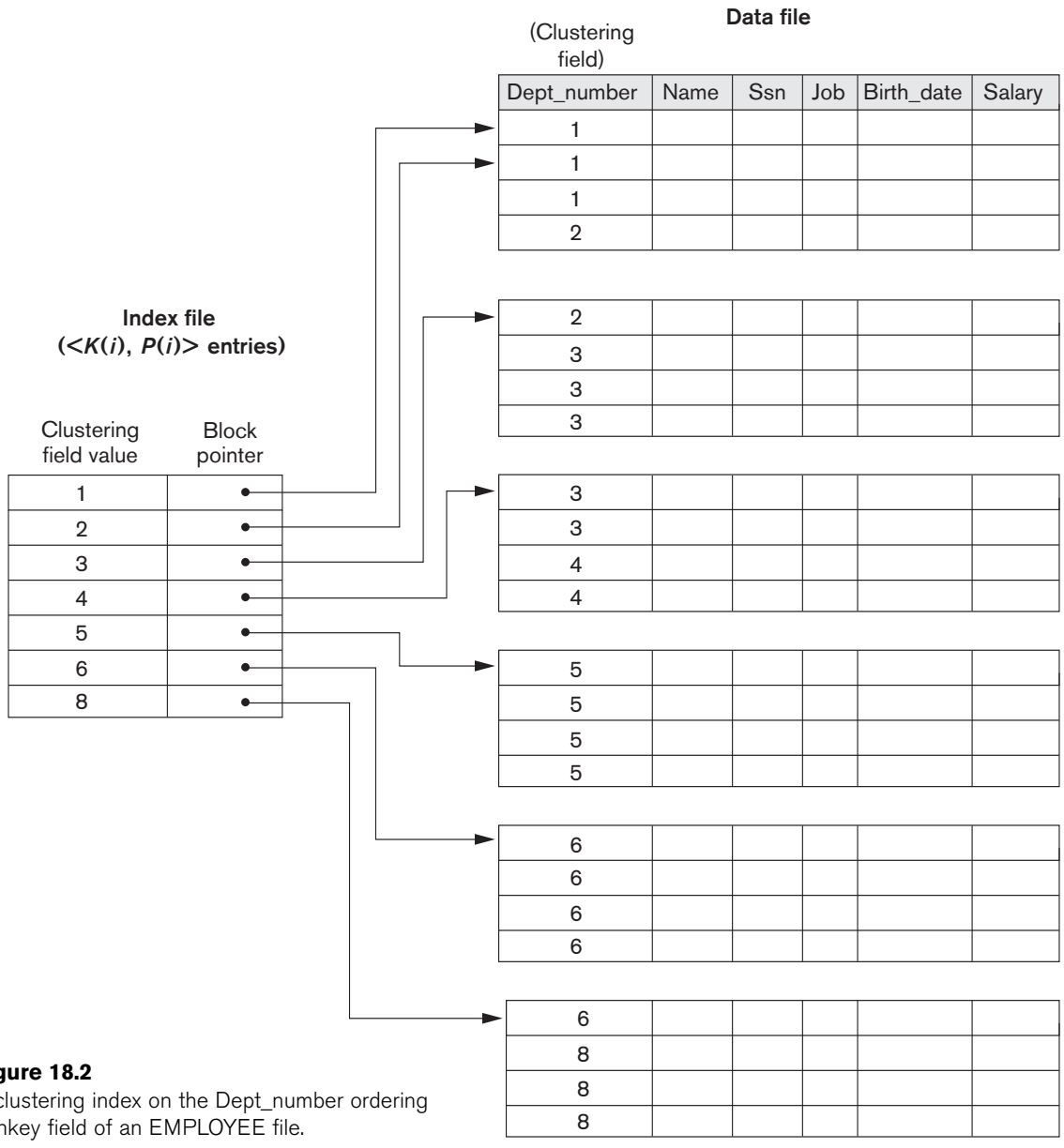
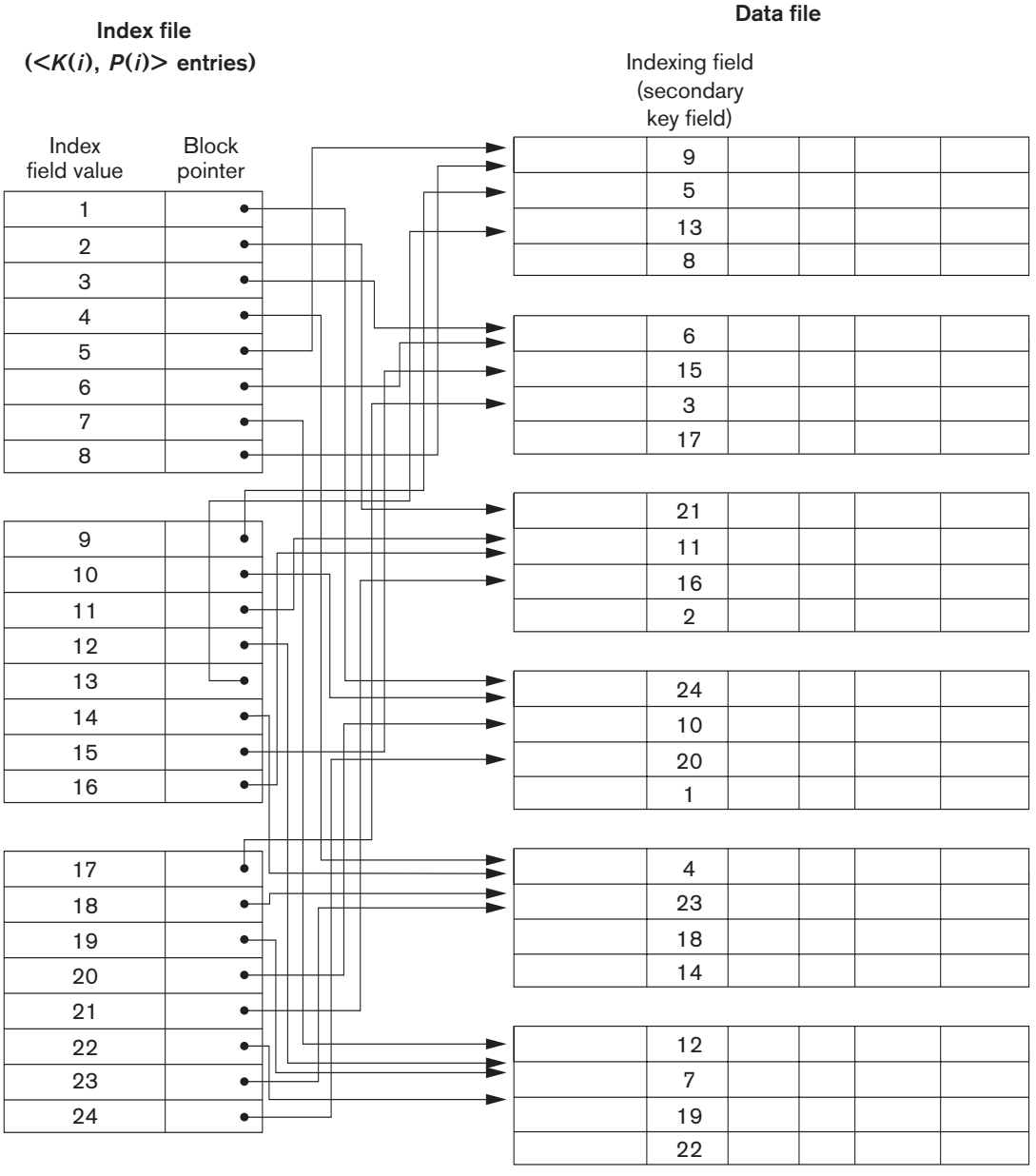


Figure 18.2
A clustering index on the Dept_number ordering nonkey field of an EMPLOYEE file.

Again we refer to the two field values of index entry i as $<K(i), P(i)>$. The entries are **ordered** by value of $K(i)$, so we can perform a binary search. Because the records of the data file are *not* physically ordered by values of the secondary key field, we *cannot* use block anchors. That is why an index entry is created for each record in the data

file, rather than for each block, as in the case of a primary index. Figure 18.4 illustrates a secondary index in which the pointers $P(i)$ in the index entries are *block pointers*, not record pointers. Once the appropriate disk block is transferred to a main memory buffer, a search for the desired record within the block can be carried out.

Figure 18.4
A dense secondary index (with block pointers) on a nonordering key field of a file.



A secondary index usually needs more storage space and longer search time than does a primary index, because of its larger number of entries. However, the *improvement* in search time for an arbitrary record is much greater for a secondary index than for a primary index, since we would have to do a *linear search* on the data file if the secondary index did not exist. For a primary index, we could still use a binary search on the main file, even if the index did not exist. Example 2 illustrates the improvement in number of blocks accessed.

Example 2. Consider the file of Example 1 with $r = 30{,}000$ fixed-length records of size $R = 100$ bytes stored on a disk with block size $B = 1024$ bytes. The file has $b = 3000$ blocks, as calculated in Example 1. Suppose we want to search for a record with a specific value for the secondary key—a nonordering key field of the file that is $V = 9$ bytes long. Without the secondary index, to do a linear search on the file would require $b/2 = 3000/2 = 1500$ block accesses on the average. Suppose that we construct a secondary index on that *nonordering key* field of the file. As in Example 1, a block pointer is $P = 6$ bytes long, so each index entry is $R_i = (9 + 6) = 15$ bytes, and the blocking factor for the index is $bfr_i = \lfloor (B/R_i) \rfloor = \lfloor (1024/15) \rfloor = 68$ entries per block. In a dense secondary index such as this, the total number of index entries r_i is equal to the *number of records* in the data file, which is 30,000. The number of blocks needed for the index is hence $b_i = \lceil (r_i/bfr_i) \rceil = \lceil (3000/68) \rceil = 442$ blocks.

A binary search on this secondary index needs $\lceil (\log_2 b_i) \rceil = \lceil (\log_2 442) \rceil = 9$ block accesses. To search for a record using the index, we need an additional block access to the data file for a total of $9 + 1 = 10$ block accesses—a vast improvement over the 1500 block accesses needed on the average for a linear search, but slightly worse than the 7 block accesses required for the primary index. This difference arose because the primary index was nondense and hence shorter, with only 45 blocks in length.

We can also create a secondary index on a *nonkey, nonordering field* of a file. In this case, numerous records in the data file can have the same value for the indexing field. There are several options for implementing such an index:

- Option 1 is to include duplicate index entries with the same $K(i)$ value—one for each record. This would be a dense index.
- Option 2 is to have variable-length records for the index entries, with a repeating field for the pointer. We keep a list of pointers $<P(i, 1), \ldots, P(i, k)>$ in the index entry for $K(i)$—one pointer to each block that contains a record whose indexing field value equals $K(i)$. In either option 1 or option 2, the binary search algorithm on the index must be modified appropriately to account for a variable number of index entries per index key value.
- Option 3, which is more commonly used, is to keep the index entries themselves at a fixed length and have a single entry for each *index field value*, but to create *an extra level of indirection* to handle the multiple pointers. In this nondense scheme, the pointer $P(i)$ in index entry $<K(i), P(i)>$ points to a disk block, which contains a *set of record pointers*; each record pointer in that disk block points to one of the data file records with value $K(i)$ for the indexing field. If some value $K(i)$ occurs in too many records, so that their record pointers cannot fit in a single disk block, a cluster or linked list of blocks is

used. This technique is illustrated in Figure 18.5. Retrieval via the index requires one or more additional block accesses because of the extra level, but the algorithms for searching the index and (more importantly) for inserting of new records in the data file are straightforward. In addition, retrievals on complex selection conditions may be handled by referring to the record pointers, without having to retrieve many unnecessary records from the data file (see Exercise 18.23).

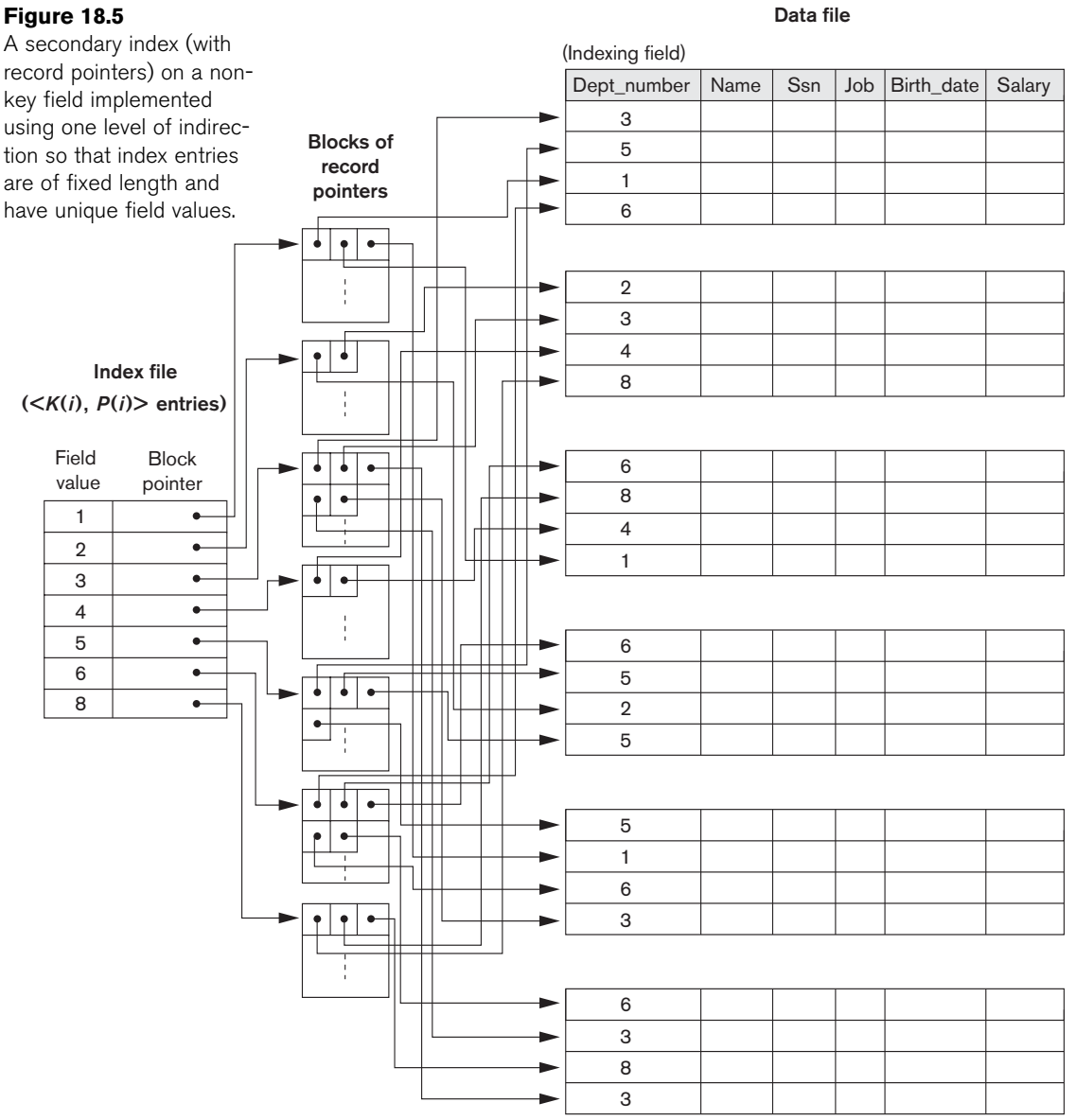


Figure 18.5
A secondary index (with record pointers) on a non-key field implemented using one level of indirection so that index entries are of fixed length and have unique field values.

Notice that a secondary index provides a **logical ordering** on the records by the indexing field. If we access the records in order of the entries in the secondary index, we get them in order of the indexing field. The primary and clustering indexes assume that the field used for **physical ordering** of records in the file is the same as the indexing field.

18.1.4 Summary

To conclude this section, we summarize the discussion of index types in two tables. Table 18.1 shows the index field characteristics of each type of ordered single-level index discussed—primary, clustering, and secondary. Table 18.2 summarizes the properties of each type of index by comparing the number of index entries and specifying which indexes are dense and which use block anchors of the data file.

Table 18.1 Types of Indexes Based on the Properties of the Indexing Field

	Index Field Used for Physical Ordering of the File	Index Field Not Used for Physical Ordering of the File
Indexing field is key	Primary index	Secondary index (Key)
Indexing field is nonkey	Clustering index	Secondary index (NonKey)

Table 18.2 Properties of Index Types

Type of Index	Number of (First-level) Index Entries	Dense or Nondense (Sparse)	Block Anchoring on the Data File
Primary	Number of blocks in data file	Nondense	Yes
Clustering	Number of distinct index field values	Nondense	Yes/no[a]
Secondary (key)	Number of records in data file	Dense	No
Secondary (nonkey)	Number of records[b] or number of distinct index field values[c]	Dense or Nondense	No

[a]Yes if every distinct value of the ordering field starts a new block; no otherwise.
[b]For option 1.
[c]For options 2 and 3.

18.2 Multilevel Indexes

The indexing schemes we have described thus far involve an ordered index file. A binary search is applied to the index to locate pointers to a disk block or to a record (or records) in the file having a specific index field value. A binary search requires approximately ($\log_2 b_i$) block accesses for an index with b_i blocks because each step of the algorithm reduces the part of the index file that we continue to search by a factor of 2. This is why we take the log function to the base 2. The idea behind a **multilevel index** is to reduce the part of the index that we continue to search by bfr_i, the blocking factor for the index, which is larger than 2. Hence, the search space is reduced much faster. The value bfr_i is called the **fan-out** of the multilevel index, and we will refer to it by the symbol ***fo***. Whereas we divide the *record search space* into two halves at each step during a binary search, we divide it n-ways (where n = the fan-out) at each search step using the multilevel index. Searching a multilevel index requires approximately ($\log_{fo} b_i$) block accesses, which is a substantially smaller number than for a binary search if the fan-out is larger than 2. In most cases, the fan-out is much larger than 2.

A multilevel index considers the index file, which we will now refer to as the **first** (or **base**) **level** of a multilevel index, as an *ordered file* with a *distinct value* for each $K(i)$. Therefore, by considering the first-level index file as a sorted data file, we can create a primary index for the first level; this index to the first level is called the **second level** of the multilevel index. Because the second level is a primary index, we can use block anchors so that the second level has one entry for *each block* of the first level. The blocking factor bfr_i for the second level—and for all subsequent levels—is the same as that for the first-level index because all index entries are the same size; each has one field value and one block address. If the first level has r_1 entries, and the blocking factor—which is also the fan-out—for the index is $bfr_i = fo$, then the first level needs $\lceil (r_1/fo) \rceil$ blocks, which is therefore the number of entries r_2 needed at the second level of the index.

We can repeat this process for the second level. The **third level**, which is a primary index for the second level, has an entry for each second-level block, so the number of third-level entries is $r_3 = \lceil (r_2/fo) \rceil$. Notice that we require a second level only if the first level needs more than one block of disk storage, and, similarly, we require a third level only if the second level needs more than one block. We can repeat the preceding process until all the entries of some index level t fit in a single block. This block at the tth level is called the **top** index level.[4] Each level reduces the number of entries at the previous level by a factor of fo—the index fan-out—so we can use the formula $1 \leq (r_1/((fo)^t))$ to calculate t. Hence, a multilevel index with r_1 first-level entries will have approximately t levels, where $t = \lceil (\log_{fo}(r_1)) \rceil$. When searching the

[4]The numbering scheme for index levels used here is the reverse of the way levels are commonly defined for tree data structures. In tree data structures, t is referred to as level 0 (zero), $t - 1$ is level 1, and so on.

index, a single disk block is retrieved at each level. Hence, t disk blocks are accessed for an index search, where t is the *number of index levels*.

The multilevel scheme described here can be used on any type of index—whether it is primary, clustering, or secondary—as long as the first-level index has *distinct values for K(i) and fixed-length entries.* Figure 18.6 shows a multilevel index built over a primary index. Example 3 illustrates the improvement in number of blocks accessed when a multilevel index is used to search for a record.

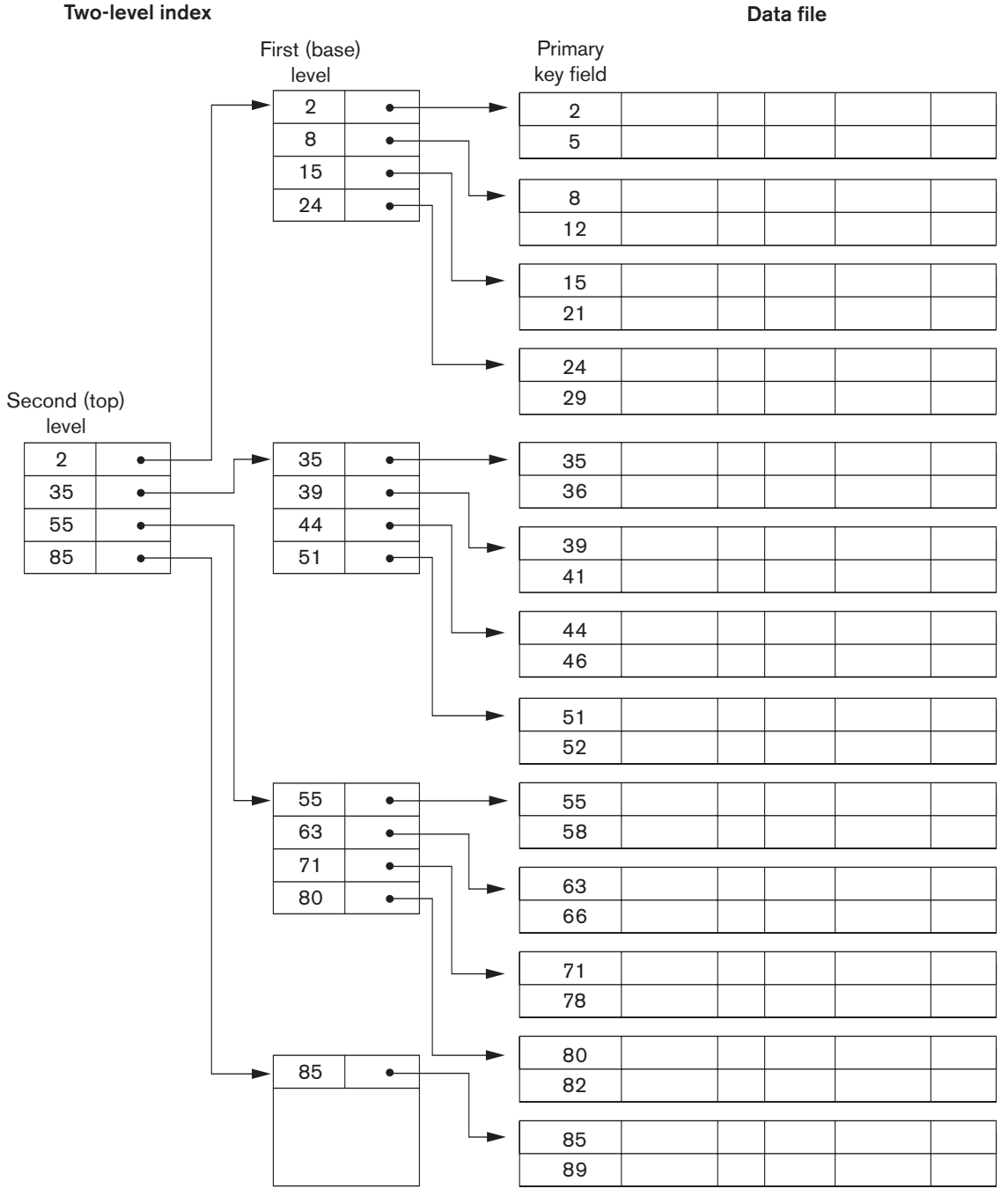
Example 3. Suppose that the dense secondary index of Example 2 is converted into a multilevel index. We calculated the index blocking factor $bfr_i = 68$ index entries per block, which is also the fan-out fo for the multilevel index; the number of first-level blocks $b_1 = 442$ blocks was also calculated. The number of second-level blocks will be $b_2 = \lceil (b_1/fo) \rceil = \lceil (442/68) \rceil = 7$ blocks, and the number of third-level blocks will be $b_3 = \lceil (b_2/fo) \rceil = \lceil (7/68) \rceil = 1$ block. Hence, the third level is the top level of the index, and $t = 3$. To access a record by searching the multilevel index, we must access one block at each level plus one block from the data file, so we need $t + 1 = 3 + 1 = 4$ block accesses. Compare this to Example 2, where 10 block accesses were needed when a single-level index and binary search were used.

Notice that we could also have a multilevel primary index, which would be nondense. Exercise 18.18(c) illustrates this case, where we *must* access the data block from the file before we can determine whether the record being searched for is in the file. For a dense index, this can be determined by accessing the first index level (without having to access a data block), since there is an index entry for *every* record in the file.

A common file organization used in business data processing is an ordered file with a multilevel primary index on its ordering key field. Such an organization is called an **indexed sequential file** and was used in a large number of early IBM systems. IBM's **ISAM** organization incorporates a two-level index that is closely related to the organization of the disk in terms of cylinders and tracks (see Section 17.2.1). The first level is a cylinder index, which has the key value of an anchor record for each cylinder of a disk pack occupied by the file and a pointer to the track index for the cylinder. The track index has the key value of an anchor record for each track in the cylinder and a pointer to the track. The track can then be searched sequentially for the desired record or block. Insertion is handled by some form of overflow file that is merged periodically with the data file. The index is recreated during file reorganization.

Algorithm 18.1 outlines the search procedure for a record in a data file that uses a nondense multilevel primary index with t levels. We refer to entry i at level j of the index as $<K_j(i), P_j(i)>$, and we search for a record whose primary key value is K. We assume that any overflow records are ignored. If the record is in the file, there must be some entry at level 1 with $K_1(i) \leq K < K_1(i + 1)$ and the record will be in the block of the data file whose address is $P_1(i)$. Exercise 18.23 discusses modifying the search algorithm for other types of indexes.

Figure 18.6
A two-level primary index resembling ISAM (Indexed Sequential Access Method) organization.



Algorithm 18.1. Searching a Nondense Multilevel Primary Index with t Levels

```
(* We assume the index entry to be a block anchor that is the first key per block. *)
p ← address of top-level block of index;
for j ← t step – 1 to 1 do
    begin
        read the index block (at jth index level) whose address is p;
        search block p for entry i such that K_j (i) ≤ K < K_j(i + 1)
     (* if K_j(i)
        is the last entry in the block, it is sufficient to satisfy K_j(i) ≤ K *);
        p ← P_j(i ) (* picks appropriate pointer at jth index level *)
    end;
    read the data file block whose address is p;
    search block p for record with key = K;
```

As we have seen, a multilevel index reduces the number of blocks accessed when searching for a record, given its indexing field value. We are still faced with the problems of dealing with index insertions and deletions, because all index levels are *physically ordered files.* To retain the benefits of using multilevel indexing while reducing index insertion and deletion problems, designers adopted a multilevel index called a **dynamic multilevel index** that leaves some space in each of its blocks for inserting new entries and uses appropriate insertion/deletion algorithms for creating and deleting new index blocks when the data file grows and shrinks. It is often implemented by using data structures called B-trees and B^+-trees, which we describe in the next section.

18.3 Dynamic Multilevel Indexes Using B-Trees and B^+-Trees

B-trees and B^+-trees are special cases of the well-known search data structure known as a **tree**. We briefly introduce the terminology used in discussing tree data structures. A **tree** is formed of **nodes**. Each node in the tree, except for a special node called the **root**, has one **parent** node and zero or more **child** nodes. The root node has no parent. A node that does not have any child nodes is called a **leaf** node; a nonleaf node is called an **internal** node. The **level** of a node is always one more than the level of its parent, with the level of the root node being *zero*.[5] A **subtree** of a node consists of that node and all its **descendant** nodes—its child nodes, the child nodes of its child nodes, and so on. A precise recursive definition of a subtree is that it consists of a node n and the subtrees of all the child nodes of n. Figure 18.7 illustrates a tree data structure. In this figure the root node is A, and its child nodes are B, C, and D. Nodes E, J, C, G, H, and K are leaf nodes. Since the leaf nodes are at different levels of the tree, this tree is called **unbalanced**.

[5]This standard definition of the level of a tree node, which we use throughout Section 18.3, is different from the one we gave for multilevel indexes in Section 18.2.

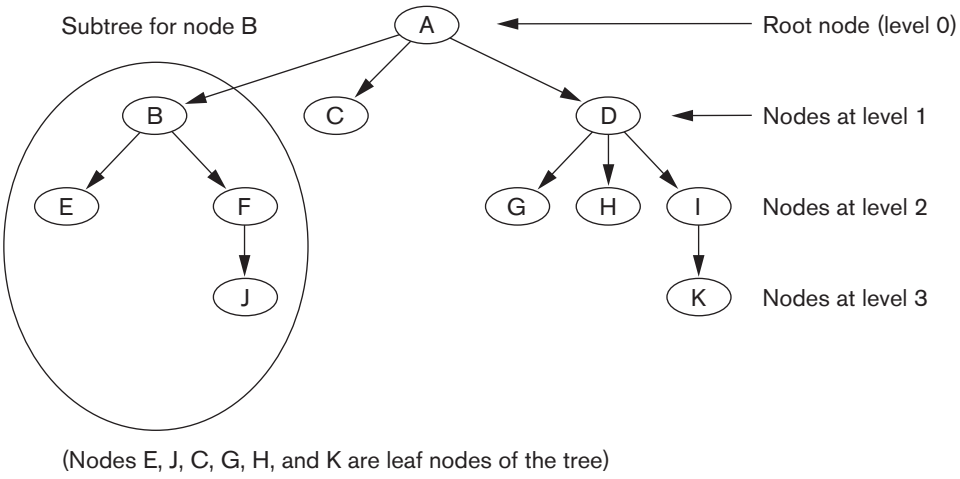


Figure 18.7
A tree data structure that shows an unbalanced tree.

In Section 18.3.1, we introduce search trees and then discuss B-trees, which can be used as dynamic multilevel indexes to guide the search for records in a data file. B-tree nodes are kept between 50 and 100 percent full, and pointers to the data blocks are stored in both internal nodes and leaf nodes of the B-tree structure. In Section 18.3.2 we discuss B+-trees, a variation of B-trees in which pointers to the data blocks of a file are stored only in leaf nodes, which can lead to fewer levels and higher-capacity indexes. In the DBMSs prevalent in the market today, the common structure used for indexing is B+-trees.

18.3.1 Search Trees and B-Trees

A **search tree** is a special type of tree that is used to guide the search for a record, given the value of one of the record's fields. The multilevel indexes discussed in Section 18.2 can be thought of as a variation of a search tree; each node in the multilevel index can have as many as *fo* pointers and *fo* key values, where *fo* is the index fan-out. The index field values in each node guide us to the next node, until we reach the data file block that contains the required records. By following a pointer, we restrict our search at each level to a subtree of the search tree and ignore all nodes not in this subtree.

Search Trees. A search tree is slightly different from a multilevel index. A **search tree of order** p is a tree such that each node contains *at most* $p - 1$ search values and p pointers in the order $<P_1, K_1, P_2, K_2, \ldots, P_{q-1}, K_{q-1}, P_q>$, where $q \leq p$. Each P_i is a pointer to a child node (or a NULL pointer), and each K_i is a search value from some

ordered set of values. All search values are assumed to be unique.[6] Figure 18.8 illustrates a node in a search tree. Two constraints must hold at all times on the search tree:

1. Within each node, $K_1 < K_2 < ... < K_{q-1}$.
2. For all values X in the subtree pointed at by P_i, we have $K_{i-1} < X < K_i$ for $1 < i < q$; $X < K_i$ for $i = 1$; and $K_{i-1} < X$ for $i = q$ (see Figure 18.8).

Whenever we search for a value X, we follow the appropriate pointer P_i according to the formulas in condition 2 above. Figure 18.9 illustrates a search tree of order $p = 3$ and integer search values. Notice that some of the pointers P_i in a node may be NULL pointers.

We can use a search tree as a mechanism to search for records stored in a disk file. The values in the tree can be the values of one of the fields of the file, called the **search field** (which is the same as the index field if a multilevel index guides the search). Each key value in the tree is associated with a pointer to the record in the data file having that value. Alternatively, the pointer could be to the disk block containing that record. The search tree itself can be stored on disk by assigning each tree node to a disk block. When a new record is inserted in the file, we must update the search tree by inserting an entry in the tree containing the search field value of the new record and a pointer to the new record.

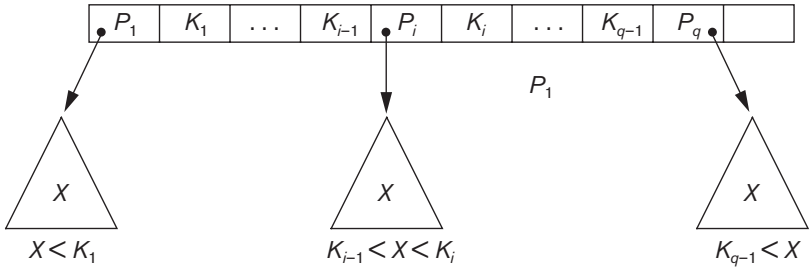


Figure 18.8
A node in a search tree with pointers to subtrees below it.

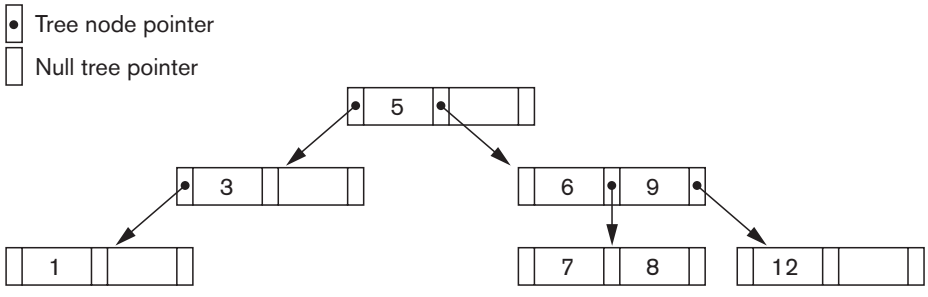


Figure 18.9
A search tree of order $p = 3$.

[6]This restriction can be relaxed. If the index is on a nonkey field, duplicate search values may exist and the node structure and the navigation rules for the tree may be modified.

Algorithms are necessary for inserting and deleting search values into and from the search tree while maintaining the preceding two constraints. In general, these algorithms do not guarantee that a search tree is **balanced**, meaning that all of its leaf nodes are at the same level.[7] The tree in Figure 18.7 is not balanced because it has leaf nodes at levels 1, 2, and 3. The goals for balancing a search tree are as follows:

- To guarantee that nodes are evenly distributed, so that the depth of the tree is minimized for the given set of keys and that the tree does not get skewed with some nodes being at very deep levels
- To make the search speed uniform, so that the average time to find any random key is roughly the same

While minimizing the number of levels in the tree is one goal, another implicit goal is to make sure that the index tree does not need too much restructuring as records are inserted into and deleted from the main file. Thus we want the nodes to be as full as possible and do not want any nodes to be empty if there are too many deletions. Record deletion may leave some nodes in the tree nearly empty, thus wasting storage space and increasing the number of levels. The B-tree addresses both of these problems by specifying additional constraints on the search tree.

B-Trees. The B-tree has additional constraints that ensure that the tree is always balanced and that the space wasted by deletion, if any, never becomes excessive. The algorithms for insertion and deletion, though, become more complex in order to maintain these constraints. Nonetheless, most insertions and deletions are simple processes; they become complicated only under special circumstances—namely, whenever we attempt an insertion into a node that is already full or a deletion from a node that makes it less than half full. More formally, a **B-tree of order *p***, when used as an access structure on a *key field* to search for records in a data file, can be defined as follows:

1. Each internal node in the B-tree (Figure 18.10(a)) is of the form
   $$<P_1, <K_1, Pr_1>, P_2, <K_2, Pr_2>, ..., <K_{q-1}, Pr_{q-1}>, P_q>$$
   where $q \leq p$. Each P_i is a **tree pointer**—a pointer to another node in the B-tree. Each Pr_i is a **data pointer**[8]—a pointer to the record whose search key field value is equal to K_i (or to the data file block containing that record).
2. Within each node, $K_1 < K_2 < ... < K_{q-1}$.
3. For all search key field values X in the subtree pointed at by P_i (the ith subtree, see Figure 18.10(a)), we have:
   $$K_{i-1} < X < K_i \text{ for } 1 < i < q; X < K_i \text{ for } i = 1; \text{ and } K_{i-1} < X \text{ for } i = q.$$
4. Each node has at most p tree pointers.

[7]The definition of *balanced* is different for binary trees. Balanced binary trees are known as *AVL trees*.

[8]A data pointer is either a block address or a record address; the latter is essentially a block address and a record offset within the block.

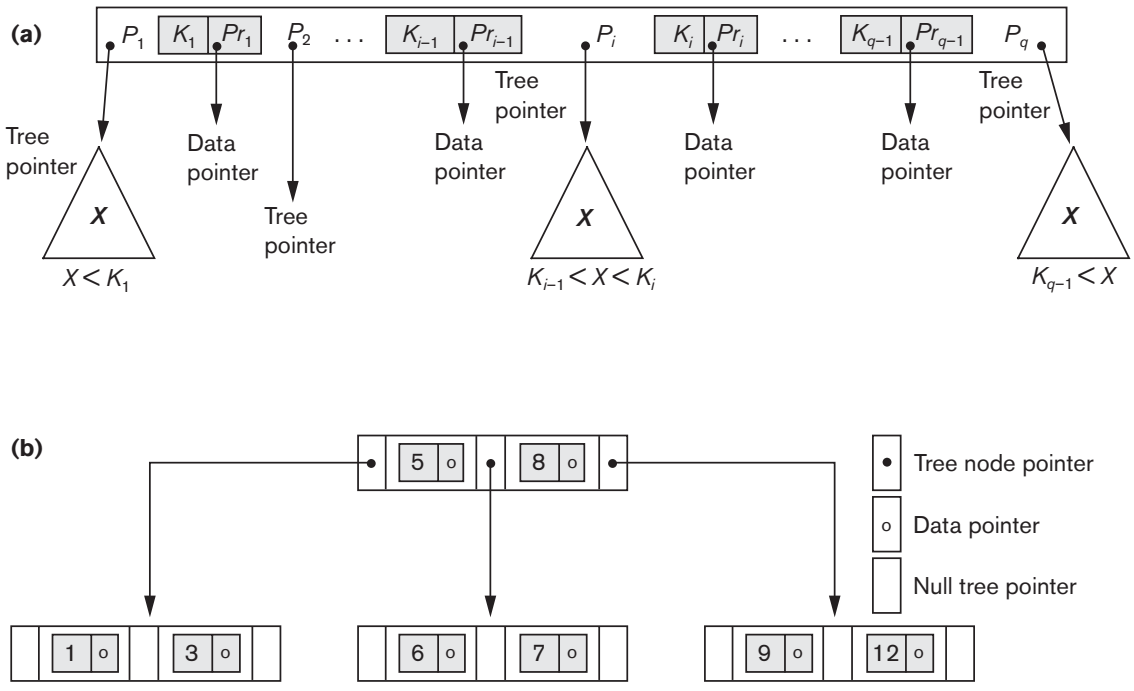


Figure 18.10
B-tree structures. (a) A node in a B-tree with $q - 1$ search values. (b) A B-tree of order $p = 3$.The values were inserted in the order 8, 5, 1, 7, 3, 12, 9, 6.

5. Each node, except the root and leaf nodes, has at least $\lceil (p/2) \rceil$ tree pointers. The root node has at least two tree pointers unless it is the only node in the tree.
6. A node with q tree pointers, $q \leq p$, has $q - 1$ search key field values (and hence has $q - 1$ data pointers).
7. All leaf nodes are at the same level. Leaf nodes have the same structure as internal nodes except that all of their *tree pointers* P_i are NULL.

Figure 18.10(b) illustrates a B-tree of order $p = 3$. Notice that all search values K in the B-tree are unique because we assumed that the tree is used as an access structure on a key field. If we use a B-tree *on a nonkey field*, we must change the definition of the file pointers Pr_i to point to a block—or a cluster of blocks—that contain the pointers to the file records. This extra level of indirection is similar to option 3, discussed in Section 18.1.3, for secondary indexes.

A B-tree starts with a single root node (which is also a leaf node) at level 0 (zero). Once the root node is full with $p - 1$ search key values and we attempt to insert another entry in the tree, the root node splits into two nodes at level 1. Only the middle value is kept in the root node, and the rest of the values are split evenly

between the other two nodes. When a nonroot node is full and a new entry is inserted into it, that node is split into two nodes at the same level, and the middle entry is moved to the parent node along with two pointers to the new split nodes. If the parent node is full, it is also split. Splitting can propagate all the way to the root node, creating a new level if the root is split. We do not discuss algorithms for B-trees in detail in this book,[9] but we outline search and insertion procedures for B^+-trees in the next section.

If deletion of a value causes a node to be less than half full, it is combined with its neighboring nodes, and this can also propagate all the way to the root. Hence, deletion can reduce the number of tree levels. It has been shown by analysis and simulation that, after numerous random insertions and deletions on a B-tree, the nodes are approximately 69 percent full when the number of values in the tree stabilizes. This is also true of B^+-trees. If this happens, node splitting and combining will occur only rarely, so insertion and deletion become quite efficient. If the number of values grows, the tree will expand without a problem—although splitting of nodes may occur, so some insertions will take more time. Each B-tree node can have *at most p* tree pointers, $p - 1$ data pointers, and $p - 1$ search key field values (see Figure 18.10(a)).

In general, a B-tree node may contain additional information needed by the algorithms that manipulate the tree, such as the number of entries q in the node and a pointer to the parent node. Next, we illustrate how to calculate the number of blocks and levels for a B-tree.

Example 4. Suppose that the search field is a nonordering key field, and we construct a B-tree on this field with $p = 23$. Assume that each node of the B-tree is 69 percent full. Each node, on the average, will have $p * 0.69 = 23 * 0.69$ or approximately 16 pointers and, hence, 15 search key field values. The **average fan-out** $fo = 16$. We can start at the root and see how many values and pointers can exist, on the average, at each subsequent level:

Root:	1 node	15 key entries	16 pointers
Level 1:	16 nodes	240 key entries	256 pointers
Level 2:	256 nodes	3840 key entries	4096 pointers
Level 3:	4096 nodes	61,440 key entries	

At each level, we calculated the number of key entries by multiplying the total number of pointers at the previous level by 15, the average number of entries in each node. Hence, for the given block size, pointer size, and search key field size, a two-level B-tree holds $3840 + 240 + 15 = 4095$ entries on the average; a three-level B-tree holds 65,535 entries on the average.

B-trees are sometimes used as **primary file organizations**. In this case, *whole records* are stored within the B-tree nodes rather than just the <search key, record pointer> entries. This works well for files with a relatively *small number of records* and a *small*

[9]For details on insertion and deletion algorithms for B-trees, consult Ramakrishnan and Gehrke [2003].

record size. Otherwise, the fan-out and the number of levels become too great to permit efficient access.

In summary, B-trees provide a multilevel access structure that is a balanced tree structure in which each node is at least half full. Each node in a B-tree of order p can have at most $p - 1$ search values.

18.3.2 B+-Trees

Most implementations of a dynamic multilevel index use a variation of the B-tree data structure called a **B+-tree**. In a B-tree, every value of the search field appears once at some level in the tree, along with a data pointer. In a B+-tree, data pointers are stored *only at the leaf nodes* of the tree; hence, the structure of leaf nodes differs from the structure of internal nodes. The leaf nodes have an entry for *every* value of the search field, along with a data pointer to the record (or to the block that contains this record) if the search field is a key field. For a nonkey search field, the pointer points to a block containing pointers to the data file records, creating an extra level of indirection.

The leaf nodes of the B+-tree are usually linked to provide ordered access on the search field to the records. These leaf nodes are similar to the first (base) level of an index. Internal nodes of the B+-tree correspond to the other levels of a multilevel index. Some search field values from the leaf nodes are *repeated* in the internal nodes of the B+-tree to guide the search. The structure of the *internal nodes* of a B+-tree of order p (Figure 18.11(a)) is as follows:

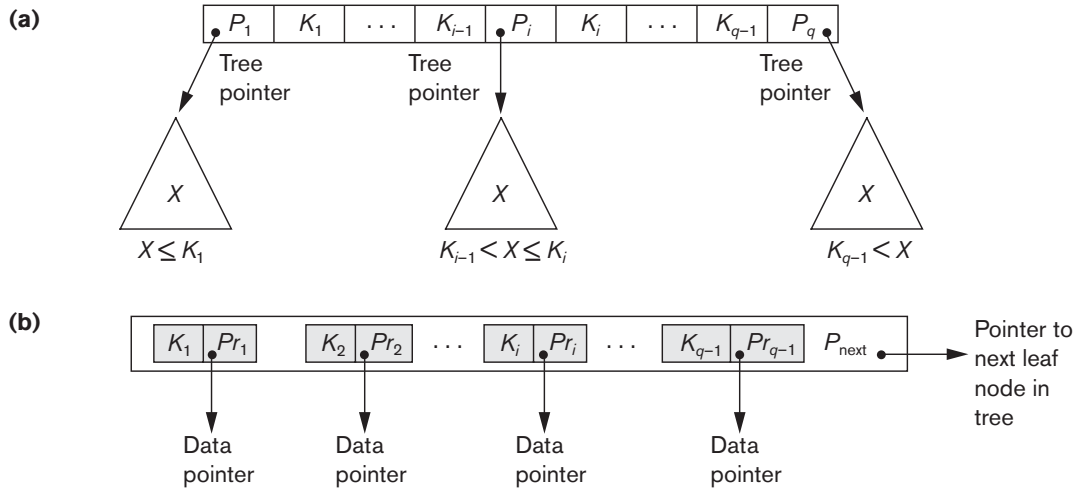
1. Each internal node is of the form
   $$<P_1, K_1, P_2, K_2, ..., P_{q-1}, K_{q-1}, P_q>$$
   where $q \leq p$ and each P_i is a **tree pointer**.
2. Within each internal node, $K_1 < K_2 < ... < K_{q-1}$.
3. For all search field values X in the subtree pointed at by P_i, we have $K_{i-1} < X \leq K_i$ for $1 < i < q$; $X \leq K_i$ for $i = 1$; and $K_{i-1} < X$ for $i = q$ (see Figure 18.11(a)).[10]
4. Each internal node has at most p tree pointers.
5. Each internal node, except the root, has at least $\lceil (p/2) \rceil$ tree pointers. The root node has at least two tree pointers if it is an internal node.
6. An internal node with q pointers, $q \leq p$, has $q - 1$ search field values.

The structure of the *leaf nodes* of a B+-tree of order p (Figure 18.11(b)) is as follows:

1. Each leaf node is of the form
   $$<<K_1, Pr_1>, <K_2, Pr_2>, ..., <K_{q-1}, Pr_{q-1}>, P_{next}>$$
   where $q \leq p$, each Pr_i is a data pointer, and P_{next} points to the next *leaf node* of the B+-tree.

[10]Our definition follows Knuth (1998). One can define a B+-tree differently by exchanging the $<$ and $\leq$ symbols ($K_{i-1} \leq X < K_i$; $K_{q-1} \leq X$), but the principles remain the same.

Figure 18.11
The nodes of a B+-tree. (a) Internal node of a B+-tree with $q - 1$ search values. (b) Leaf node of a B+-tree with $q - 1$ search values and $q - 1$ data pointers.



2. Within each leaf node, $K_1 \leq K_2 \ldots, K_{q-1}, q \leq p$.
3. Each Pr_i is a **data pointer** that points to the record whose search field value is K_i or to a file block containing the record (or to a block of record pointers that point to records whose search field value is K_i if the search field is not a key).
4. Each leaf node has at least $\lceil (p/2) \rceil$ values.
5. All leaf nodes are at the same level.

The pointers in internal nodes are *tree pointers* to blocks that are tree nodes, whereas the pointers in leaf nodes are *data pointers* to the data file records or blocks—except for the P_{next} pointer, which is a tree pointer to the next leaf node. By starting at the leftmost leaf node, it is possible to traverse leaf nodes as a linked list, using the P_{next} pointers. This provides ordered access to the data records on the indexing field. A P_{previous} pointer can also be included. For a B+-tree on a nonkey field, an extra level of indirection is needed similar to the one shown in Figure 18.5, so the *Pr* pointers are block pointers to blocks that contain a set of record pointers to the actual records in the data file, as discussed in option 3 of Section 18.1.3.

Because entries in the *internal nodes* of a B+-tree include search values and tree pointers without any data pointers, more entries can be packed into an internal node of a B+-tree than for a similar B-tree. Thus, for the same block (node) size, the order p will be larger for the B+-tree than for the B-tree, as we illustrate in Example 5. This can lead to fewer B+-tree levels, improving search time. Because the structures for internal and for leaf nodes of a B+-tree are different, the order p can be different. We

will use p to denote the order for *internal nodes* and p_{leaf} to denote the order for *leaf nodes*, which we define as being the maximum number of data pointers in a leaf node.

Example 5. To calculate the order p of a B^+-tree, suppose that the search key field is $V = 9$ bytes long, the block size is $B = 512$ bytes, a record pointer is $Pr = 7$ bytes, and a block pointer is $P = 6$ bytes. An internal node of the B^+-tree can have up to p tree pointers and $p - 1$ search field values; these must fit into a single block. Hence, we have:

$$(p * P) + ((p - 1) * V) \leq B$$
$$(P * 6) + ((P - 1) * 9) \leq 512$$
$$(15 * p) \leq 521$$

We can choose p to be the largest value satisfying the above inequality, which gives $p = 34$. This is larger than the value of 23 for the B-tree (it is left to the reader to compute the order of the B-tree assuming same size pointers), resulting in a larger fan-out and more entries in each internal node of a B^+-tree than in the corresponding B-tree. The leaf nodes of the B^+-tree will have the same number of values and pointers, except that the pointers are data pointers and a next pointer. Hence, the order p_{leaf} for the leaf nodes can be calculated as follows:

$$(p_{leaf} * (Pr + V)) + P \leq B$$
$$(p_{leaf} * (7 + 9)) + 6 \leq 512$$
$$(16 * p_{leaf}) \leq 506$$

It follows that each leaf node can hold up to $p_{leaf} = 31$ key value/data pointer combinations, assuming that the data pointers are record pointers.

As with the B-tree, we may need additional information—to implement the insertion and deletion algorithms—in each node. This information can include the type of node (internal or leaf), the number of current entries q in the node, and pointers to the parent and sibling nodes. Hence, before we do the above calculations for p and p_{leaf}, we should reduce the block size by the amount of space needed for all such information. The next example illustrates how we can calculate the number of entries in a B^+-tree.

Example 6. Suppose that we construct a B^+-tree on the field in Example 5. To calculate the approximate number of entries in the B^+-tree, we assume that each node is 69 percent full. On the average, each internal node will have $34 * 0.69$ or approximately 23 pointers, and hence 22 values. Each leaf node, on the average, will hold $0.69 * p_{leaf} = 0.69 * 31$ or approximately 21 data record pointers. A B^+-tree will have the following average number of entries at each level:

Root:	1 node	22 key entries	23 pointers
Level 1:	23 nodes	506 key entries	529 pointers
Level 2:	529 nodes	11,638 key entries	12,167 pointers
Leaf level:	12,167 nodes	255,507 data record pointers	

For the block size, pointer size, and search field size given above, a three-level B+-tree holds up to 255,507 record pointers, with the average 69 percent occupancy of nodes. Compare this to the 65,535 entries for the corresponding B-tree in Example 4. This is the main reason that B+-trees are preferred to B-trees as indexes to database files.

Search, Insertion, and Deletion with B+-Trees. Algorithm 18.2 outlines the procedure using the B+-tree as the access structure to search for a record. Algorithm 18.3 illustrates the procedure for inserting a record in a file with a B+-tree access structure. These algorithms assume the existence of a key search field, and they must be modified appropriately for the case of a B+-tree on a nonkey field. We illustrate insertion and deletion with an example.

Algorithm 18.2. Searching for a Record with Search Key Field Value K, Using a B+-tree

```
n ← block containing root node of B+-tree;
read block n;
while (n is not a leaf node of the B+-tree) do
   begin
   q ← number of tree pointers in node n;
   if K ≤ n.K1 (*n.Ki refers to the ith search field value in node n*)
      then n ← n.P1 (*n.Pi refers to the ith tree pointer in node n*)
      else if K > n.Kq–1
         then n ← n.Pq
         else begin
            search node n for an entry i such that n.Ki–1 < K ≤ n.Ki;
            n ← n.Pi
            end;
   read block n
   end;
search block n for entry (Ki, Pri) with K = Ki; (* search leaf node *)
if found
   then read data file block with address Pri and retrieve record
   else the record with search field value K is not in the data file;
```

Algorithm 18.3. Inserting a Record with Search Key Field Value K in a B+-tree of Order p

```
n ← block containing root node of B+-tree;
read block n; set stack S to empty;
while (n is not a leaf node of the B+-tree) do
   begin
   push address of n on stack S;
      (*stack S holds parent nodes that are needed in case of split*)
   q ← number of tree pointers in node n;
   if K ≤ n.K1 (*n.Ki refers to the ith search field value in node n*)
```

```
        then n ← n.P1 (*n.Pi refers to the ith tree pointer in node n*)
        else if K > n.Kq–1
            then n ← n.Pq
            else begin
                search node n for an entry i such that n.Ki–1 < K ≤ n.Ki;
                n ← n.Pi
                end;
        read block n
    end;
search block n for entry (Ki,Pri) with K = Ki; (*search leaf node n*)
if found
    then record already in file; cannot insert
    else (*insert entry in B+-tree to point to record*)
        begin
        create entry (K, Pr) where Pr points to the new record;
        if leaf node n is not full
            then insert entry (K, Pr) in correct position in leaf node n
            else begin (*leaf node n is full with pleaf record pointers; is split*)
                copy n to temp (*temp is an oversize leaf node to hold extra
                                    entries*);
                insert entry (K, Pr) in temp in correct position;
                (*temp now holds pleaf + 1 entries of the form (Ki, Pri)*)
                new ← a new empty leaf node for the tree; new.Pnext ← n.Pnext ;
                j ← ⌈(pleaf + 1)/2⌉;
                n ← first j entries in temp (up to entry (Kj, Prj)); n.Pnext ← new;
                new ← remaining entries in temp; K ← Kj ;
                (*now we must move (K, new) and insert in parent internal node;
                  however, if parent is full, split may propagate*)
                finished ← false;
                repeat
                if stack S is empty
                    then (*no parent node; new root node is created for the tree*)
                        begin
                        root ← a new empty internal node for the tree;
                        root ← <n, K, new>; finished ← true;
                        end
                    else begin
                        n ← pop stack S;
                        if internal node n is not full
                            then
                                begin (*parent node not full; no split*)
                                insert (K, new) in correct position in internal node n;
                                finished ← true
                                end
                            else begin (*internal node n is full with p tree pointers;
                                        overflow condition; node is split*)
```

```
                copy n to temp (*temp is an oversize internal node*);
                insert (K, new) in temp in correct position;
                (*temp now has p + 1 tree pointers*)
                new ← a new empty internal node for the tree;
                j ← ⌊((p + 1)/2)⌋;
                n ← entries up to tree pointer P_j in temp;
                (*n contains <P_1, K_1, P_2, K_2, ..., P_{j-1}, K_{j-1}, P_j >*)
                new ← entries from tree pointer P_{j+1} in temp;
                (*new contains < P_{j+1}, K_{j+1}, ..., K_{p-1}, P_p, K_p, P_{p+1} >*)
                K ← K_j
                (*now we must move (K, new) and insert in parent
                  internal node*)
                end
            end
        until finished
        end;
end;
```

Figure 18.12 illustrates insertion of records in a B+-tree of order $p = 3$ and $p_{leaf} = 2$. First, we observe that the root is the only node in the tree, so it is also a leaf node. As soon as more than one level is created, the tree is divided into internal nodes and leaf nodes. Notice that *every key value must exist at the leaf level*, because all data pointers are at the leaf level. However, only some values exist in internal nodes to guide the search. Notice also that every value appearing in an internal node also appears as *the rightmost value* in the leaf level of the subtree pointed at by the tree pointer to the left of the value.

When a *leaf node* is full and a new entry is inserted there, the node *overflows* and must be split. The first $j = \lceil ((p_{leaf} + 1)/2) \rceil$ entries in the original node are kept there, and the remaining entries are moved to a new leaf node. The jth search value is replicated in the parent internal node, and an extra pointer to the new node is created in the parent. These must be inserted in the parent node in their correct sequence. If the parent internal node is full, the new value will cause it to overflow also, so it must be split. The entries in the internal node up to P_j—the jth tree pointer after inserting the new value and pointer, where $j = \lfloor ((p + 1)/2) \rfloor$—are kept, while the jth search value is moved to the parent, not replicated. A new internal node will hold the entries from P_{j+1} to the end of the entries in the node (see Algorithm 18.3). This splitting can propagate all the way up to create a new root node and hence a new level for the B+-tree.

Figure 18.13 illustrates deletion from a B+-tree. When an entry is deleted, it is always removed from the leaf level. If it happens to occur in an internal node, it must also be removed from there. In the latter case, the value to its left in the leaf node must replace it in the internal node because that value is now the rightmost entry in the subtree. Deletion may cause **underflow** by reducing the number of entries in the leaf node to below the minimum required. In this case, we try to find a sibling leaf node—a leaf node directly to the left or to the right of the node with underflow—

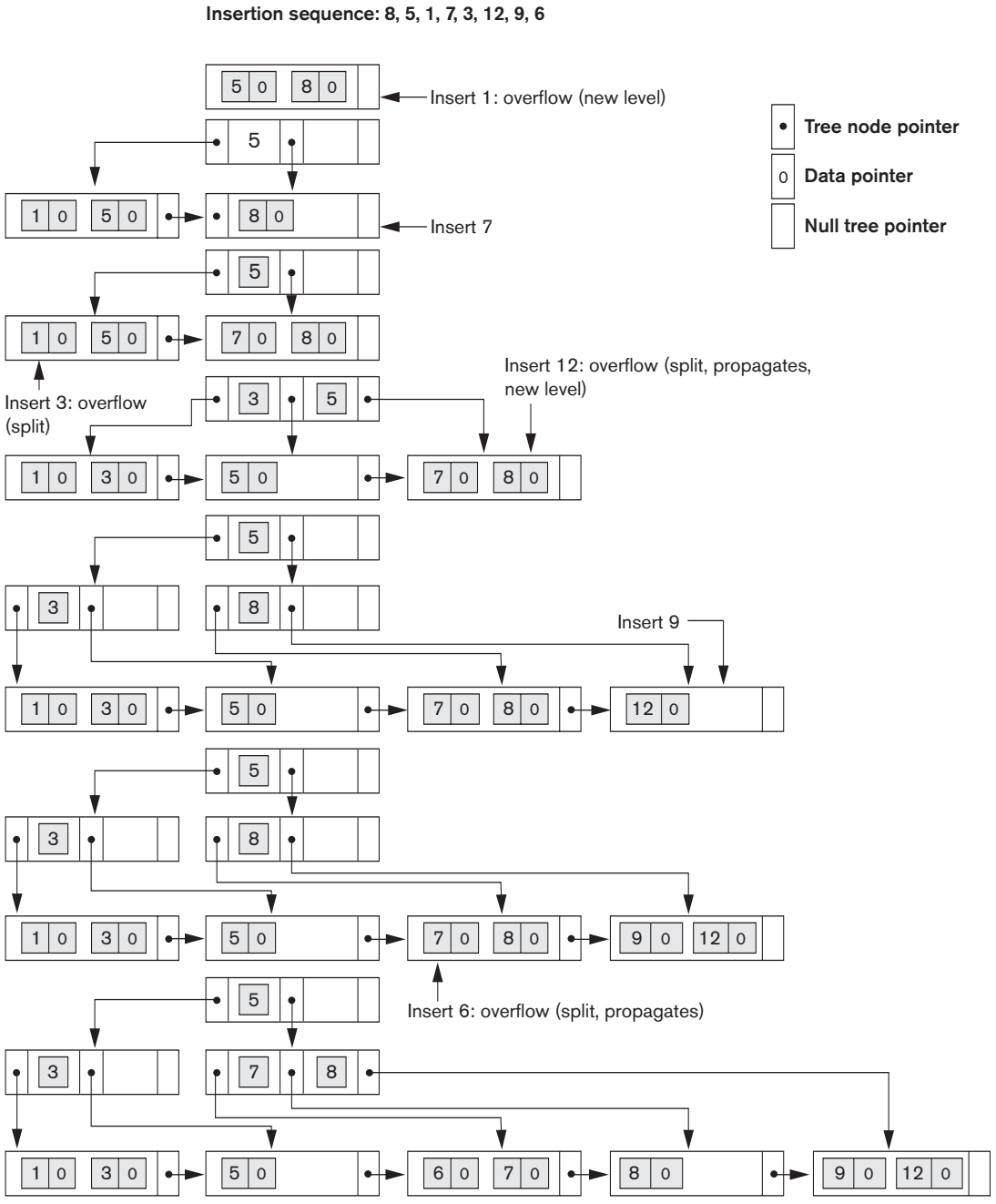


Figure 18.12
An example of insertion in a B$^+$-tree with $p = 3$ and $p_{leaf} = 2$.

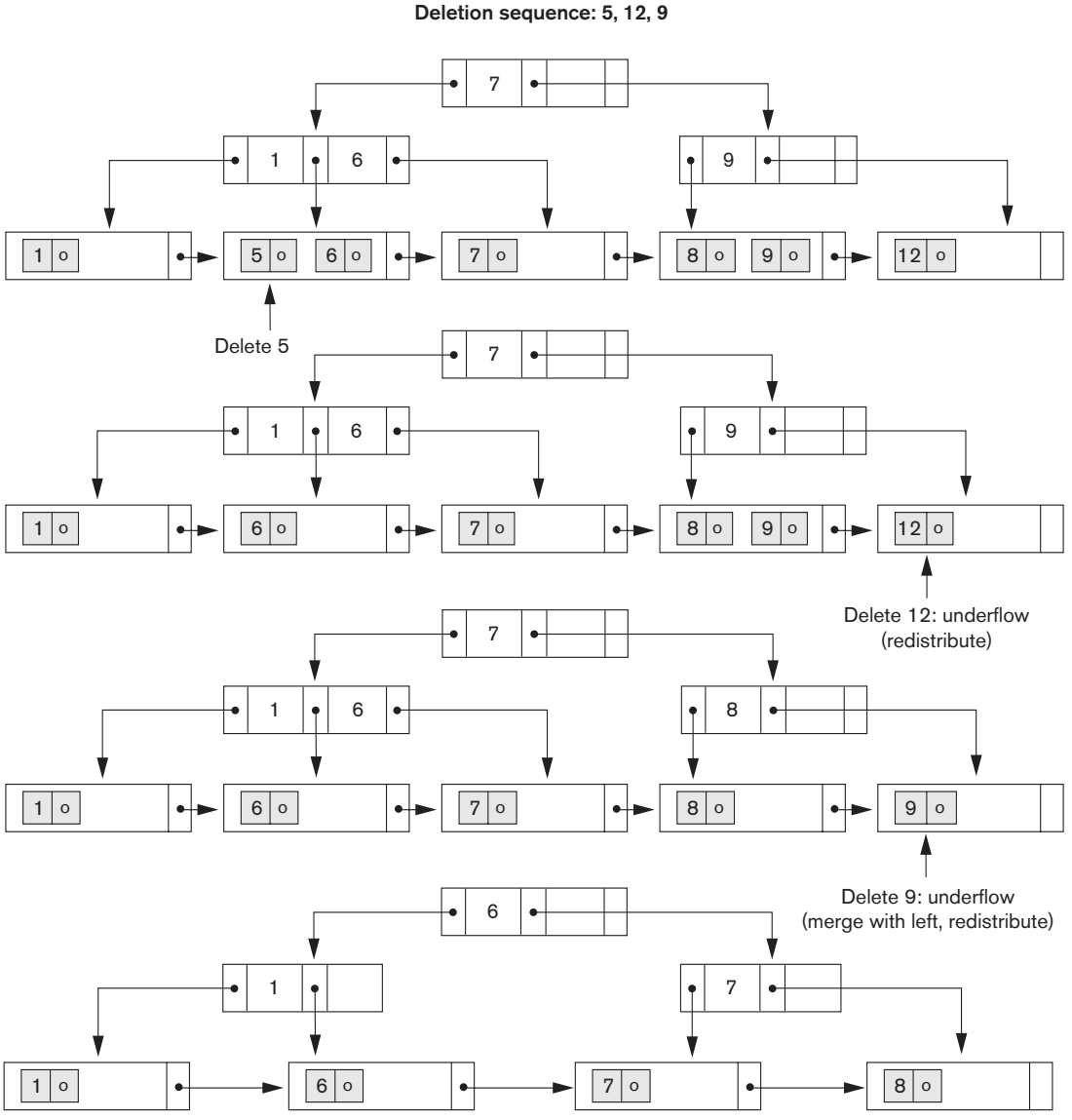


Figure 18.13
An example of deletion from a B+-tree.

and redistribute the entries among the node and its **sibling** so that both are at least half full; otherwise, the node is merged with its siblings and the number of leaf nodes is reduced. A common method is to try to **redistribute** entries with the left sibling; if this is not possible, an attempt to redistribute with the right sibling is

made. If this is also not possible, the three nodes are merged into two leaf nodes. In such a case, underflow may propagate to **internal** nodes because one fewer tree pointer and search value are needed. This can propagate and reduce the tree levels.

Notice that implementing the insertion and deletion algorithms may require parent and sibling pointers for each node, or the use of a stack as in Algorithm 18.3. Each node should also include the number of entries in it and its type (leaf or internal). Another alternative is to implement insertion and deletion as recursive procedures.[11]

Variations of B-Trees and B$^+$-Trees. To conclude this section, we briefly mention some variations of B-trees and B$^+$-trees. In some cases, constraint 5 on the B-tree (or for the internal nodes of the B$^+$–tree, except the root node), which requires each node to be at least half full, can be changed to require each node to be at least two-thirds full. In this case the B-tree has been called a **B*-tree**. In general, some systems allow the user to choose a **fill factor** between 0.5 and 1.0, where the latter means that the B-tree (index) nodes are to be completely full. It is also possible to specify two fill factors for a B$^+$-tree: one for the leaf level and one for the internal nodes of the tree. When the index is first constructed, each node is filled up to approximately the fill factors specified. Some investigators have suggested relaxing the requirement that a node be half full, and instead allow a node to become completely empty before merging, to simplify the deletion algorithm. Simulation studies show that this does not waste too much additional space under randomly distributed insertions and deletions.

18.4 Indexes on Multiple Keys

In our discussion so far, we have assumed that the primary or secondary keys on which files were accessed were single attributes (fields). In many retrieval and update requests, multiple attributes are involved. If a certain combination of attributes is used frequently, it is advantageous to set up an access structure to provide efficient access by a key value that is a combination of those attributes.

For example, consider an EMPLOYEE file containing attributes Dno (department number), Age, Street, City, Zip_code, Salary and Skill_code, with the key of Ssn (Social Security number). Consider the query: *List the employees in department number 4 whose age is 59*. Note that both Dno and Age are nonkey attributes, which means that a search value for either of these will point to multiple records. The following alternative search strategies may be considered:

1. Assuming Dno has an index, but Age does not, access the records having Dno = 4 using the index, and then select from among them those records that satisfy Age = 59.

[11]For more details on insertion and deletion algorithms for B$^+$ trees, consult Ramakrishnan and Gehrke [2003].

2. Alternately, if Age is indexed but Dno is not, access the records having Age = 59 using the index, and then select from among them those records that satisfy Dno = 4.
3. If indexes have been created on both Dno and Age, both indexes may be used; each gives a set of records or a set of pointers (to blocks or records). An intersection of these sets of records or pointers yields those records or pointers that satisfy both conditions.

All of these alternatives eventually give the correct result. However, if the set of records that meet each condition (Dno = 4 or Age = 59) individually are large, yet only a few records satisfy the combined condition, then none of the above is an efficient technique for the given search request. A number of possibilities exist that would treat the combination < Dno, Age> or < Age, Dno> as a search key made up of multiple attributes. We briefly outline these techniques in the following sections. We will refer to keys containing multiple attributes as **composite keys**.

18.4.1 Ordered Index on Multiple Attributes

All the discussion in this chapter so far still applies if we create an index on a search key field that is a combination of <Dno, Age>. The search key is a pair of values <4, 59> in the above example. In general, if an index is created on attributes $<A_1, A_2, \ldots, A_n>$, the search key values are tuples with n values: $<v_1, v_2, \ldots, v_n>$.

A lexicographic ordering of these tuple values establishes an order on this composite search key. For our example, all of the department keys for department number 3 precede those for department number 4. Thus <3, n> precedes <4, m> for any values of m and n. The ascending key order for keys with Dno = 4 would be <4, 18>, <4, 19>, <4, 20>, and so on. Lexicographic ordering works similarly to ordering of character strings. An index on a composite key of n attributes works similarly to any index discussed in this chapter so far.

18.4.2 Partitioned Hashing

Partitioned hashing is an extension of static external hashing (Section 17.8.2) that allows access on multiple keys. It is suitable only for equality comparisons; range queries are not supported. In partitioned hashing, for a key consisting of n components, the hash function is designed to produce a result with n separate hash addresses. The bucket address is a concatenation of these n addresses. It is then possible to search for the required composite search key by looking up the appropriate buckets that match the parts of the address in which we are interested.

For example, consider the composite search key <Dno, Age>. If Dno and Age are hashed into a 3-bit and 5-bit address respectively, we get an 8-bit bucket address. Suppose that Dno = 4 has a hash address '100' and Age = 59 has hash address '10101'. Then to search for the combined search value, Dno = 4 and Age = 59, one goes to bucket address 100 10101; just to search for all employees with Age = 59, all buckets (eight of them) will be searched whose addresses are '000 10101', '001 10101', ... and

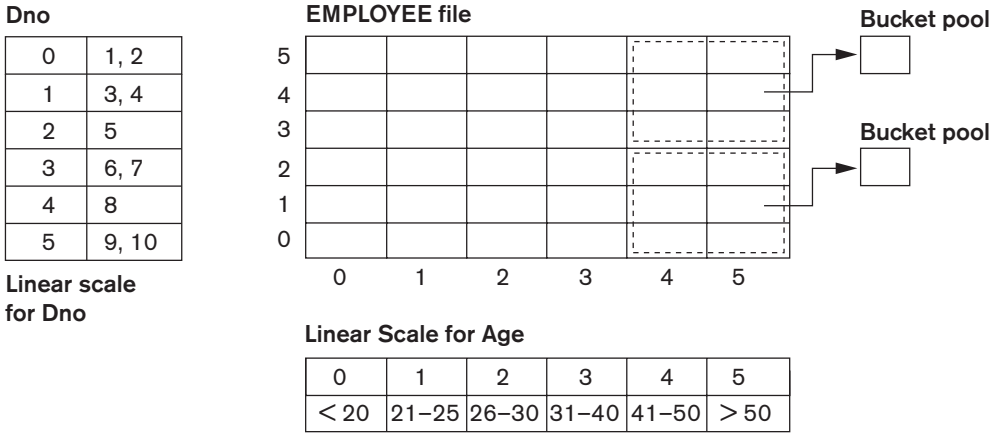
so on. An advantage of partitioned hashing is that it can be easily extended to any number of attributes. The bucket addresses can be designed so that high-order bits in the addresses correspond to more frequently accessed attributes. Additionally, no separate access structure needs to be maintained for the individual attributes. The main drawback of partitioned hashing is that it cannot handle range queries on any of the component attributes.

18.4.3 Grid Files

Another alternative is to organize the EMPLOYEE file as a grid file. If we want to access a file on two keys, say Dno and Age as in our example, we can construct a grid array with one linear scale (or dimension) for each of the search attributes. Figure 18.14 shows a grid array for the EMPLOYEE file with one linear scale for Dno and another for the Age attribute. The scales are made in a way as to achieve a uniform distribution of that attribute. Thus, in our example, we show that the linear scale for Dno has Dno = 1, 2 combined as one value 0 on the scale, while Dno = 5 corresponds to the value 2 on that scale. Similarly, Age is divided into its scale of 0 to 5 by grouping ages so as to distribute the employees uniformly by age. The grid array shown for this file has a total of 36 cells. Each cell points to some bucket address where the records corresponding to that cell are stored. Figure 18.14 also shows the assignment of cells to buckets (only partially).

Thus our request for Dno = 4 and Age = 59 maps into the cell (1, 5) corresponding to the grid array. The records for this combination will be found in the corresponding bucket. This method is particularly useful for range queries that would map into a set of cells corresponding to a group of values along the linear scales. If a range query corresponds to a match on the some of the grid cells, it can be processed by accessing exactly the buckets for those grid cells. For example, a query for Dno ≤ 5

Figure 18.14
Example of a grid array on Dno and Age attributes.



and Age > 40 refers to the data in the top bucket shown in Figure 18.14. The grid file concept can be applied to any number of search keys. For example, for *n* search keys, the grid array would have *n* dimensions. The grid array thus allows a partitioning of the file along the dimensions of the search key attributes and provides an access by combinations of values along those dimensions. Grid files perform well in terms of reduction in time for multiple key access. However, they represent a space overhead in terms of the grid array structure. Moreover, with dynamic files, a frequent reorganization of the file adds to the maintenance cost.[12]

18.5 Other Types of Indexes

18.5.1 Hash Indexes

It is also possible to create access structures similar to indexes that are based on *hashing*. The **hash index** is a secondary structure to access the file by using hashing on a search key other than the one used for the primary data file organization. The index entries are of the type <*K*, *Pr*> or <*K*, *P*>, where *Pr* is a pointer to the record containing the key, or *P* is a pointer to the block containing the record for that key. The index file with these index entries can be organized as a dynamically expandable hash file, using one of the techniques described in Section 17.8.3; searching for an entry uses the hash search algorithm on *K*. Once an entry is found, the pointer *Pr* (or *P*) is used to locate the corresponding record in the data file. Figure 18.15 illustrates a hash index on the Emp_id field for a file that has been stored as a sequential file ordered by Name. The Emp_id is hashed to a bucket number by using a hashing function: the sum of the digits of Emp_id modulo 10. For example, to find Emp_id 51024, the hash function results in bucket number 2; that bucket is accessed first. It contains the index entry < 51024, *Pr* >; the pointer *Pr* leads us to the actual record in the file. In a practical application, there may be thousands of buckets; the bucket number, which may be several bits long, would be subjected to the directory schemes discussed about dynamic hashing in Section 17.8.3. Other search structures can also be used as indexes.

18.5.2 Bitmap Indexes

The **bitmap index** is another popular data structure that facilitates querying on multiple keys. Bitmap indexing is used for relations that contain a large number of rows. It creates an index for one or more columns, and each value or value range in those columns is indexed. Typically, a bitmap index is created for those columns that contain a fairly small number of unique values. To build a bitmap index on a set of records in a relation, the records must be numbered from 0 to *n* with an id (a record id or a row id) that can be mapped to a physical address made of a block number and a record offset within the block.

[12]Insertion/deletion algorithms for grid files may be found in Nievergelt et al. (1984).

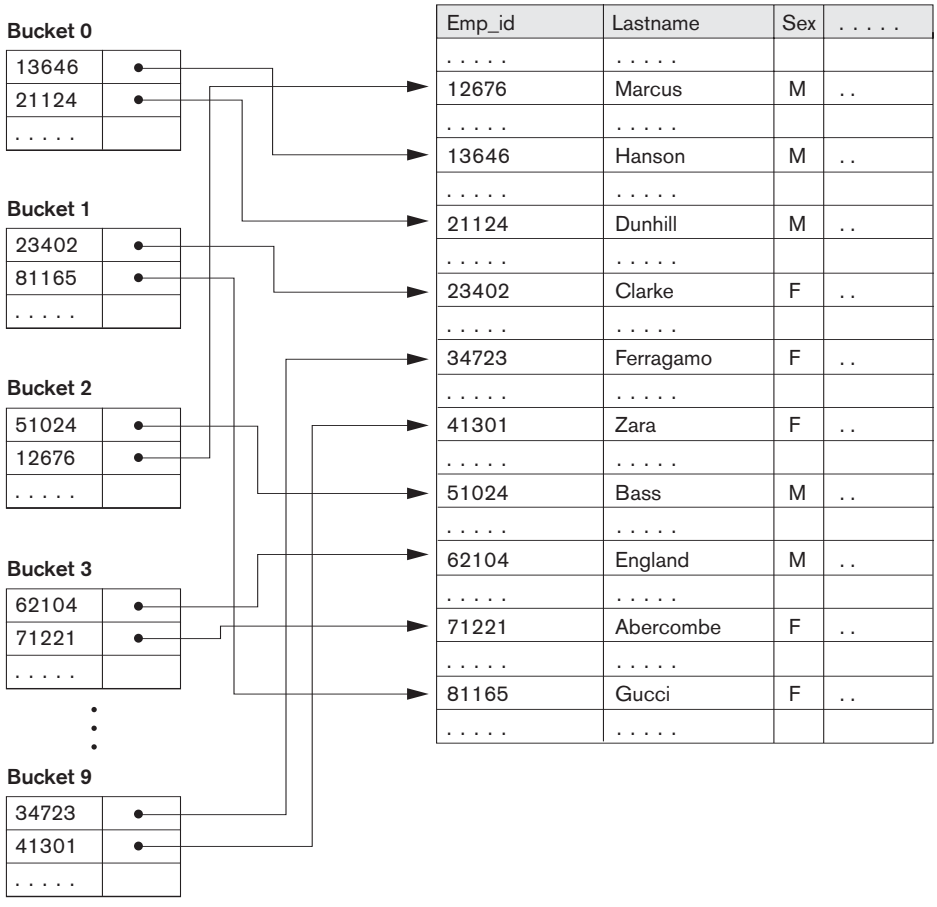


Figure 18.15
Hash-based indexing.

A bitmap index is built on one particular value of a particular field (the column in a relation) and is just an array of bits. Consider a bitmap index for the column C and a value V for that column. For a relation with n rows, it contains n bits. The i^{th} bit is set to 1 if the row i has the value V for column C; otherwise it is set to a 0. If C contains the valueset $<v_1, v_2, ..., v_m>$ with m distinct values, then m bitmap indexes would be created for that column. Figure 18.16 shows the relation EMPLOYEE with columns Emp_id, Lname, Sex, Zipcode, and Salary_grade (with just 8 rows for illustration) and a bitmap index for the Sex and Zipcode columns. As an example, if the bitmap for Sex = F, the bits for Row_ids 1, 3, 4, and 7 are set to 1, and the rest of the bits are set to 0, the bitmap indexes could have the following query applications:

- For the query $C_1 = V_1$, the corresponding bitmap for value V_1 returns the Row_ids containing the rows that qualify.

EMPLOYEE

Row_id	Emp_id	Lname	Sex	Zipcode	Salary_grade
0	51024	Bass	M	94040	..
1	23402	Clarke	F	30022	..
2	62104	England	M	19046	..
3	34723	Ferragamo	F	30022	..
4	81165	Gucci	F	19046	..
5	13646	Hanson	M	19046	..
6	12676	Marcus	M	30022	..
7	41301	Zara	F	94040	..

Bitmap index for Sex

M	F
10100110	01011001

Bitmap index for Zipcode

Zipcode 19046	Zipcode 30022	Zipcode 94040
00101100	01010010	10000001

Figure 18.16
Bitmap indexes for Sex and Zipcode

- For the query $C_1 = V_1$ and $C_2 = V_2$ (a multikey search request), the two corresponding bitmaps are retrieved and intersected (logically AND-ed) to yield the set of Row_ids that qualify. In general, k bitvectors can be intersected to deal with k equality conditions. Complex AND-OR conditions can also be supported using bitmap indexing.
- To retrieve a count of rows that qualify for the condition $C_1 = V_1$, the "1" entries in the corresponding bitvector are counted.
- Queries with negation, such as $C_1 \neg = V_1$, can be handled by applying the Boolean *complement* operation on the corresponding bitmap.

Consider the example in Figure 18.16. To find employees with Sex = F and Zipcode = 30022, we intersect the bitmaps "01011001" and "01010010" yielding Row_ids 1 and 3. Employees who do not live in Zipcode = 94040 are obtained by complementing the bitvector "10000001" and yields Row_ids 1 through 6. In general, if we assume uniform distribution of values for a given column, and if one column has 5 distinct values and another has 10 distinct values, the join condition on these two can be considered to have a selectivity of 1/50 (=1/5 * 1/10). Hence, only about 2 percent of the records would actually have to be retrieved. If a column has only a few values, like the Sex column in Figure 18.16, retrieval of the Sex = M condition on average would retrieve 50 percent of the rows; in such cases, it is better to do a complete scan rather than use bitmap indexing.

In general, bitmap indexes are efficient in terms of the storage space that they need. If we consider a file of 1 million rows (records) with record size of 100 bytes per row, each bitmap index would take up only one bit per row and hence would use 1 million bits or 125 Kbytes. Suppose this relation is for 1 million residents of a state, and they are spread over 200 ZIP Codes; the 200 bitmaps over Zipcodes contribute 200 bits (or 25 bytes) worth of space per row; hence, the 200 bitmaps occupy only 25 percent as much space as the data file. They allow an exact retrieval of all residents who live in a given ZIP Code by yielding their Row_ids.

When records are deleted, renumbering rows and shifting bits in bitmaps becomes expensive. Another bitmap, called the **existence bitmap**, can be used to avoid this expense. This bitmap has a 0 bit for the rows that have been deleted but are still present and a 1 bit for rows that actually exist. Whenever a row is inserted in the relation, an entry must be made in all the bitmaps of all the columns that have a bitmap index; rows typically are appended to the relation or may replace deleted rows. This process represents an indexing overhead.

Large bitvectors are handled by treating them as a series of 32-bit or 64-bit vectors, and corresponding AND, OR, and NOT operators are used from the instruction set to deal with 32- or 64-bit input vectors in a single instruction. This makes bitvector operations computationally very efficient.

Bitmaps for B^+-Tree Leaf Nodes. Bitmaps can be used on the leaf nodes of B^+-tree indexes as well as to point to the set of records that contain each specific value of the indexed field in the leaf node. When the B^+-tree is built on a nonkey search field, the leaf record must contain a list of record pointers alongside each value of the indexed attribute. For values that occur very frequently, that is, in a large percentage of the relation, a bitmap index may be stored instead of the pointers. As an example, for a relation with n rows, suppose a value occurs in 10 percent of the file records. A bitvector would have n bits, having the "1" bit for those Row_ids that contain that search value, which is $n/8$ or $0.125n$ bytes in size. If the record pointer takes up 4 bytes (32 bits), then the $n/10$ record pointers would take up $4 * n/10$ or $0.4n$ bytes. Since $0.4n$ is more than 3 times larger than $0.125n$, it is better to store the bitmap index rather than the record pointers. Hence for search values that occur more frequently than a certain ratio (in this case that would be 1/32), it is beneficial to use bitmaps as a compressed storage mechanism for representing the record pointers in B^+-trees that index a nonkey field.

18.5.3 Function-Based Indexing

In this section we discuss a new type of indexing, called **function-based indexing**, that has been introduced in the Oracle relational DBMS as well as in some other commercial products.[13]

The idea behind function-based indexing is to create an index such that the value that results from applying some function on a field or a collection of fields becomes the key to the index. The following examples show how to create and use function-based indexes.

Example 1. The following statement creates a function-based index on the EMPLOYEE table based on an uppercase representation of the Lname column, which can be entered in many ways but is always queried by its uppercase representation.

```
CREATE INDEX upper_ix ON Employee (UPPER(Lname));
```

[13]Rafi Ahmed contributed most of this section.

This statement will create an index based on the function `UPPER(Lname)`, which returns the last name in uppercase letters; for example, `UPPER('Smith')` will return 'SMITH'.

Function-based indexes ensure that Oracle Database system will use the index rather than perform a full table scan, even when a function is used in the search predicate of a query. For example, the following query will use the index:

```
SELECT First_name, Lname
FROM Employee
WHERE UPPER(Lname)= "SMITH".
```

Without the function-based index, an Oracle Database might perform a full table scan, since a B^+-tree index is searched only by using the column value directly; the use of any function on a column prevents such an index from being used.

Example 2. In this example, the EMPLOYEE table is supposed to contain two fields—salary and commission_pct (commission percentage)—and an index is being created on the sum of salary and commission based on the commission_pct.

```
CREATE INDEX income_ix
ON Employee(Salary + (Salary*Commission_pct));
```

The following query uses the `income_ix` index even though the fields salary and commission_pct are occurring in the reverse order in the query when compared to the index definition.

```
SELECT First_name, Lname
FROM Employee
WHERE ((Salary*Commission_pct) + Salary ) > 15000;
```

Example 3. This is a more advanced example of using function-based indexing to define conditional uniqueness. The following statement creates a unique function-based index on the ORDERS table that prevents a customer from taking advantage of a promotion id ("blowout sale") more than once. It creates a composite index on the Customer_id and Promotion_id fields together, and it allows only one entry in the index for a given Customer_id with the Promotion_id of "2" by declaring it as a unique index.

```
CREATE UNIQUE INDEX promo_ix ON Orders
(CASE WHEN Promotion_id = 2 THEN Customer_id ELSE NULL END,
CASE WHEN Promotion_id = 2 THEN Promotion_id ELSE NULL END);
```

Note that by using the **`CASE`** statement, the objective is to remove from the index any rows where Promotion_id is not equal to 2. Oracle Database does not store in the B^+-tree index any rows where all the keys are NULL. Therefore, in this example, we map both Customer_id and Promotion_id to NULL unless Promotion_id is equal to 2. The result is that the index constraint is violated only if Promotion_id is equal to 2, for two (attempted insertions of) rows with the same Customer_id value.

18.6 Some General Issues Concerning Indexing

18.6.1 Logical versus Physical Indexes

In the earlier discussion, we have assumed that the index entries <*K*, *Pr*> (or <*K*, *P*>) always include a physical pointer *Pr* (or *P*) that specifies the physical record address on disk as a block number and offset. This is sometimes called a **physical index**, and it has the disadvantage that the pointer must be changed if the record is moved to another disk location. For example, suppose that a primary file organization is based on linear hashing or extendible hashing; then, each time a bucket is split, some records are allocated to new buckets and hence have new physical addresses. If there was a secondary index on the file, the pointers to those records would have to be found and updated, which is a difficult task.

To remedy this situation, we can use a structure called a **logical index**, whose index entries are of the form <K, K_p>. Each entry has one value K for the secondary indexing field matched with the value K_p of the field used for the primary file organization. By searching the secondary index on the value of K, a program can locate the corresponding value of K_p and use this to access the record through the primary file organization. Logical indexes thus introduce an additional level of indirection between the access structure and the data. They are used when physical record addresses are expected to change frequently. The cost of this indirection is the extra search based on the primary file organization.

18.6.2 Discussion

In many systems, an index is not an integral part of the data file but can be created and discarded dynamically. That is why it is often called an *access structure.* Whenever we expect to access a file frequently based on some search condition involving a particular field, we can request the DBMS to create an index on that field. Usually, a secondary index is created to avoid physical ordering of the records in the data file on disk.

The main advantage of secondary indexes is that—theoretically, at least—they can be created in conjunction with *virtually any primary record organization.* Hence, a secondary index could be used to complement other primary access methods such as ordering or hashing, or it could even be used with mixed files. To create a B^+-tree secondary index on some field of a file, we must go through all records in the file to create the entries at the leaf level of the tree. These entries are then sorted and filled according to the specified fill factor; simultaneously, the other index levels are created. It is more expensive and much harder to create primary indexes and clustering indexes dynamically, because the records of the data file must be physically sorted on disk in order of the indexing field. However, some systems allow users to create these indexes dynamically on their files by sorting the file during index creation.

It is common to use an index to enforce a *key constraint* on an attribute. While searching the index to insert a new record, it is straightforward to check at the same

time whether another record in the file—and hence in the index tree—has the same key attribute value as the new record. If so, the insertion can be rejected.

If an index is created on a nonkey field, *duplicates* occur; handling of these duplicates is an issue the DBMS product vendors have to deal with and affects data storage as well as index creation and management. Data records for the duplicate key may be contained in the same block or may span multiple blocks where many duplicates are possible. Some systems add a row id to the record so that records with duplicate keys have their own unique identifiers. In such cases, the B^+-tree index may regard a <key, Row_id> combination as the de facto key for the index, turning the index into a unique index with no duplicates. The deletion of a key *K* from such an index would involve deleting all occurrences of that key *K*—hence the deletion algorithm has to account for this.

In actual DBMS products, deletion from B^+-tree indexes is also handled in various ways to improve performance and response times. Deleted records may be marked as deleted and the corresponding index entries may also not be removed until a garbage collection process reclaims the space in the data file; the index is rebuilt online after garbage collection.

A file that has a secondary index on every one of its fields is often called a **fully inverted file**. Because all indexes are secondary, new records are inserted at the end of the file; therefore, the data file itself is an unordered (heap) file. The indexes are usually implemented as B^+-trees, so they are updated dynamically to reflect insertion or deletion of records. Some commercial DBMSs, such as Software AG's Adabas, use this method extensively.

We referred to the popular IBM file organization called ISAM in Section 18.2. Another IBM method, the **virtual storage access method** (**VSAM**), is somewhat similar to the B^+–tree access structure and is still being used in many commercial systems.

18.6.3 Column-Based Storage of Relations

There has been a recent trend to consider a column-based storage of relations as an alternative to the traditional way of storing relations row by row. Commercial relational DBMSs have offered B^+-tree indexing on primary as well as secondary keys as an efficient mechanism to support access to data by various search criteria and the ability to write a row or a set of rows to disk at a time to produce write-optimized systems. For data warehouses (to be discussed in Chapter 29), which are read-only databases, the column-based storage offers particular advantages for read-only queries. Typically, the column-store RDBMSs consider storing each column of data individually and afford performance advantages in the following areas:

- Vertically partitioning the table column by column, so that a two-column table can be constructed for every attribute and thus only the needed columns can be accessed
- Use of column-wise indexes (similar to the bitmap indexes discussed in Section 18.5.2) and join indexes on multiple tables to answer queries without having to access the data tables

- Use of materialized views (see Chapter 5) to support queries on multiple columns

Column-wise storage of data affords additional freedom in the creation of indexes, such as the bitmap indexes discussed earlier. The same column may be present in multiple projections of a table and indexes may be created on each projection. To store the values in the same column, strategies for data compression, null-value suppression, dictionary encoding techniques (where distinct values in the column are assigned shorter codes), and run-length encoding techniques have been devised. MonetDB/X100, C-Store, and Vertica are examples of such systems. Further discussion on column-store DBMSs can be found in the references mentioned in this chapter's Selected Bibliography.

18.7 Summary

In this chapter we presented file organizations that involve additional access structures, called indexes, to improve the efficiency of retrieval of records from a data file. These access structures may be used *in conjunction with* the primary file organizations discussed in Chapter 17, which are used to organize the file records themselves on disk.

Three types of ordered single-level indexes were introduced: primary, clustering, and secondary. Each index is specified on a field of the file. Primary and clustering indexes are constructed on the physical ordering field of a file, whereas secondary indexes are specified on nonordering fields as additional access structures to improve performance of queries and transactions. The field for a primary index must also be a key of the file, whereas it is a nonkey field for a clustering index. A single-level index is an ordered file and is searched using a binary search. We showed how multilevel indexes can be constructed to improve the efficiency of searching an index.

Next we showed how multilevel indexes can be implemented as B-trees and B^+-trees, which are dynamic structures that allow an index to expand and shrink dynamically. The nodes (blocks) of these index structures are kept between half full and completely full by the insertion and deletion algorithms. Nodes eventually stabilize at an average occupancy of 69 percent full, allowing space for insertions without requiring reorganization of the index for the majority of insertions. B^+-trees can generally hold more entries in their internal nodes than can B-trees, so they may have fewer levels or hold more entries than does a corresponding B-tree.

We gave an overview of multiple key access methods, and showed how an index can be constructed based on hash data structures. We discussed the **hash index** in some detail—it is a secondary structure to access the file by using hashing on a search key other than that used for the primary organization. Bitmap indexing is another important type of indexing used for querying by multiple keys and is particularly applicable on fields with a small number of unique values. Bitmaps can also be used at the leaf nodes of B^+ tree indexes as well. We also discussed function-based indexing, which is being provided by relational vendors to allow special indexes on a function of one or more attributes.

We introduced the concept of a logical index and compared it with the physical indexes we described before. They allow an additional level of indirection in indexing in order to permit greater freedom for movement of actual record locations on disk. We also reviewed some general issues related to indexing, and commented on column-based storage of relations, which has particular advantages for read-only databases. Finally, we discussed how combinations of the above organizations can be used. For example, secondary indexes are often used with mixed files, as well as with unordered and ordered files.

Review Questions

18.1. Define the following terms: *indexing field*, *primary key field*, *clustering field*, *secondary key field*, *block anchor*, *dense index*, and *nondense (sparse) index*.

18.2. What are the differences among primary, secondary, and clustering indexes? How do these differences affect the ways in which these indexes are implemented? Which of the indexes are dense, and which are not?

18.3. Why can we have at most one primary or clustering index on a file, but several secondary indexes?

18.4. How does multilevel indexing improve the efficiency of searching an index file?

18.5. What is the order p of a B-tree? Describe the structure of B-tree nodes.

18.6. What is the order p of a B^+-tree? Describe the structure of both internal and leaf nodes of a B^+-tree.

18.7. How does a B-tree differ from a B^+-tree? Why is a B^+-tree usually preferred as an access structure to a data file?

18.8. Explain what alternative choices exist for accessing a file based on multiple search keys.

18.9. What is partitioned hashing? How does it work? What are its limitations?

18.10. What is a grid file? What are its advantages and disadvantages?

18.11. Show an example of constructing a grid array on two attributes on some file.

18.12. What is a fully inverted file? What is an indexed sequential file?

18.13. How can hashing be used to construct an index?

18.14. What is bitmap indexing? Create a relation with two columns and sixteen tuples and show an example of a bitmap index on one or both.

18.15. What is the concept of function-based indexing? What additional purpose does it serve?

18.16. What is the difference between a logical index and a physical index?

18.17. What is column-based storage of a relational database?

Exercises

18.18. Consider a disk with block size $B = 512$ bytes. A block pointer is $P = 6$ bytes long, and a record pointer is $P_R = 7$ bytes long. A file has $r = 30{,}000$ EMPLOYEE records of *fixed length*. Each record has the following fields: Name (30 bytes), Ssn (9 bytes), Department_code (9 bytes), Address (40 bytes), Phone (10 bytes), Birth_date (8 bytes), Sex (1 byte), Job_code (4 bytes), and Salary (4 bytes, real number). An additional byte is used as a deletion marker.

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. Suppose that the file is *ordered* by the key field Ssn and we want to construct a *primary index* on Ssn. Calculate (i) the index blocking factor bfr_i (which is also the index fan-out *fo*); (ii) the number of first-level index entries and the number of first-level index blocks; (iii) the number of levels needed if we make it into a multilevel index; (iv) the total number of blocks required by the multilevel index; and (v) the number of block accesses needed to search for and retrieve a record from the file—given its Ssn value—using the primary index.

d. Suppose that the file is *not ordered* by the key field Ssn and we want to construct a *secondary index* on Ssn. Repeat the previous exercise (part c) for the secondary index and compare with the primary index.

e. Suppose that the file is *not ordered* by the nonkey field Department_code and we want to construct a *secondary index* on Department_code, using option 3 of Section 18.1.3, with an extra level of indirection that stores record pointers. Assume there are 1,000 distinct values of Department_code and that the EMPLOYEE records are evenly distributed among these values. Calculate (i) the index blocking factor bfr_i (which is also the index fan-out *fo*); (ii) the number of blocks needed by the level of indirection that stores record pointers; (iii) the number of first-level index entries and the number of first-level index blocks; (iv) the number of levels needed if we make it into a multilevel index; (v) the total number of blocks required by the multilevel index and the blocks used in the extra level of indirection; and (vi) the approximate number of block accesses needed to search for and retrieve all records in the file that have a specific Department_code value, using the index.

f. Suppose that the file is *ordered* by the nonkey field Department_code and we want to construct a *clustering index* on Department_code that uses block anchors (every new value of Department_code starts at the beginning of a new block). Assume there are 1,000 distinct values of Department_code and that the EMPLOYEE records are evenly distributed among these values. Calculate (i) the index blocking factor bfr_i (which is also the index fan-out *fo*); (ii) the number of first-level index entries and the number of first-level index blocks; (iii) the number of levels needed if we make it into a multilevel index; (iv) the total number of blocks

required by the multilevel index; and (v) the number of block accesses needed to search for and retrieve all records in the file that have a specific Department_code value, using the clustering index (assume that multiple blocks in a cluster are contiguous).

g. Suppose that the file is *not* ordered by the key field Ssn and we want to construct a B^+-tree access structure (index) on Ssn. Calculate (i) the orders p and p_{leaf} of the B^+-tree; (ii) the number of leaf-level blocks needed if blocks are approximately 69 percent full (rounded up for convenience); (iii) the number of levels needed if internal nodes are also 69 percent full (rounded up for convenience); (iv) the total number of blocks required by the B^+-tree; and (v) the number of block accesses needed to search for and retrieve a record from the file—given its Ssn value—using the B^+-tree.

h. Repeat part g, but for a B-tree rather than for a B^+-tree. Compare your results for the B-tree and for the B^+-tree.

18.19. A PARTS file with Part# as the key field includes records with the following Part# values: 23, 65, 37, 60, 46, 92, 48, 71, 56, 59, 18, 21, 10, 74, 78, 15, 16, 20, 24, 28, 39, 43, 47, 50, 69, 75, 8, 49, 33, 38. Suppose that the search field values are inserted in the given order in a B^+-tree of order $p = 4$ and $p_{leaf} = 3$; show how the tree will expand and what the final tree will look like.

18.20. Repeat Exercise 18.19, but use a B-tree of order $p = 4$ instead of a B^+-tree.

18.21. Suppose that the following search field values are deleted, in the given order, from the B^+-tree of Exercise 18.19; show how the tree will shrink and show the final tree. The deleted values are 65, 75, 43, 18, 20, 92, 59, 37.

18.22. Repeat Exercise 18.21, but for the B-tree of Exercise 18.20.

18.23. Algorithm 18.1 outlines the procedure for searching a nondense multilevel primary index to retrieve a file record. Adapt the algorithm for each of the following cases:

a. A multilevel secondary index on a nonkey nonordering field of a file. Assume that option 3 of Section 18.1.3 is used, where an extra level of indirection stores pointers to the individual records with the corresponding index field value.

b. A multilevel secondary index on a nonordering key field of a file.

c. A multilevel clustering index on a nonkey ordering field of a file.

18.24. Suppose that several secondary indexes exist on nonkey fields of a file, implemented using option 3 of Section 18.1.3; for example, we could have secondary indexes on the fields Department_code, Job_code, and Salary of the EMPLOYEE file of Exercise 18.18. Describe an efficient way to search for and retrieve records satisfying a complex selection condition on these fields, such as (Department_code = 5 AND Job_code = 12 AND Salary = 50,000), using the record pointers in the indirection level.

18.25. Adapt Algorithms 18.2 and 18.3, which outline search and insertion procedures for a B^+-tree, to a B-tree.

18.26. It is possible to modify the B^+-tree insertion algorithm to delay the case where a new level is produced by checking for a possible *redistribution* of values among the leaf nodes. Figure 18.17 (next page) illustrates how this could be done for our example in Figure 18.12; rather than splitting the leftmost leaf node when 12 is inserted, we do a *left redistribution* by moving 7 to the leaf node to its left (if there is space in this node). Figure 18.17 shows how the tree would look when redistribution is considered. It is also possible to consider *right redistribution*. Try to modify the B^+-tree insertion algorithm to take redistribution into account.

18.27. Outline an algorithm for deletion from a B^+-tree.

18.28. Repeat Exercise 18.27 for a B-tree.

Selected Bibliography

Bayer and McCreight (1972) introduced B-trees and associated algorithms. Comer (1979) provides an excellent survey of B-trees and their history, and variations of B-trees. Knuth (1998) provides detailed analysis of many search techniques, including B-trees and some of their variations. Nievergelt (1974) discusses the use of binary search trees for file organization. Textbooks on file structures including Claybrook (1992), Smith and Barnes (1987), and Salzberg (1988), the algorithms and data structures textbook by Wirth (1985), as well as the database textbook by Ramakrihnan and Gehrke (2003) discuss indexing in detail and may be consulted for search, insertion, and deletion algorithms for B-trees and B^+-trees. Larson (1981) analyzes index-sequential files, and Held and Stonebraker (1978) compare static multilevel indexes with B-tree dynamic indexes. Lehman and Yao (1981) and Srinivasan and Carey (1991) did further analysis of concurrent access to B-trees. The books by Wiederhold (1987), Smith and Barnes (1987), and Salzberg (1988), among others, discuss many of the search techniques described in this chapter. Grid files are introduced in Nievergelt et al. (1984). Partial-match retrieval, which uses partitioned hashing, is discussed in Burkhard (1976, 1979).

New techniques and applications of indexes and B^+-trees are discussed in Lanka and Mays (1991), Zobel et al. (1992), and Faloutsos and Jagadish (1992). Mohan and Narang (1992) discuss index creation. The performance of various B–tree and B^+-tree algorithms is assessed in Baeza-Yates and Larson (1989) and Johnson and Shasha (1993). Buffer management for indexes is discussed in Chan et al. (1992). Column-based storage of databases was proposed by Stonebraker et al. (2005) in the C-Store database system; MonetDB/X100 by Boncz et al. (2008) is another implementation of the idea. Abadi et al. (2008) discuss the advantages of column stores over row-stored databases for read-only database applications.

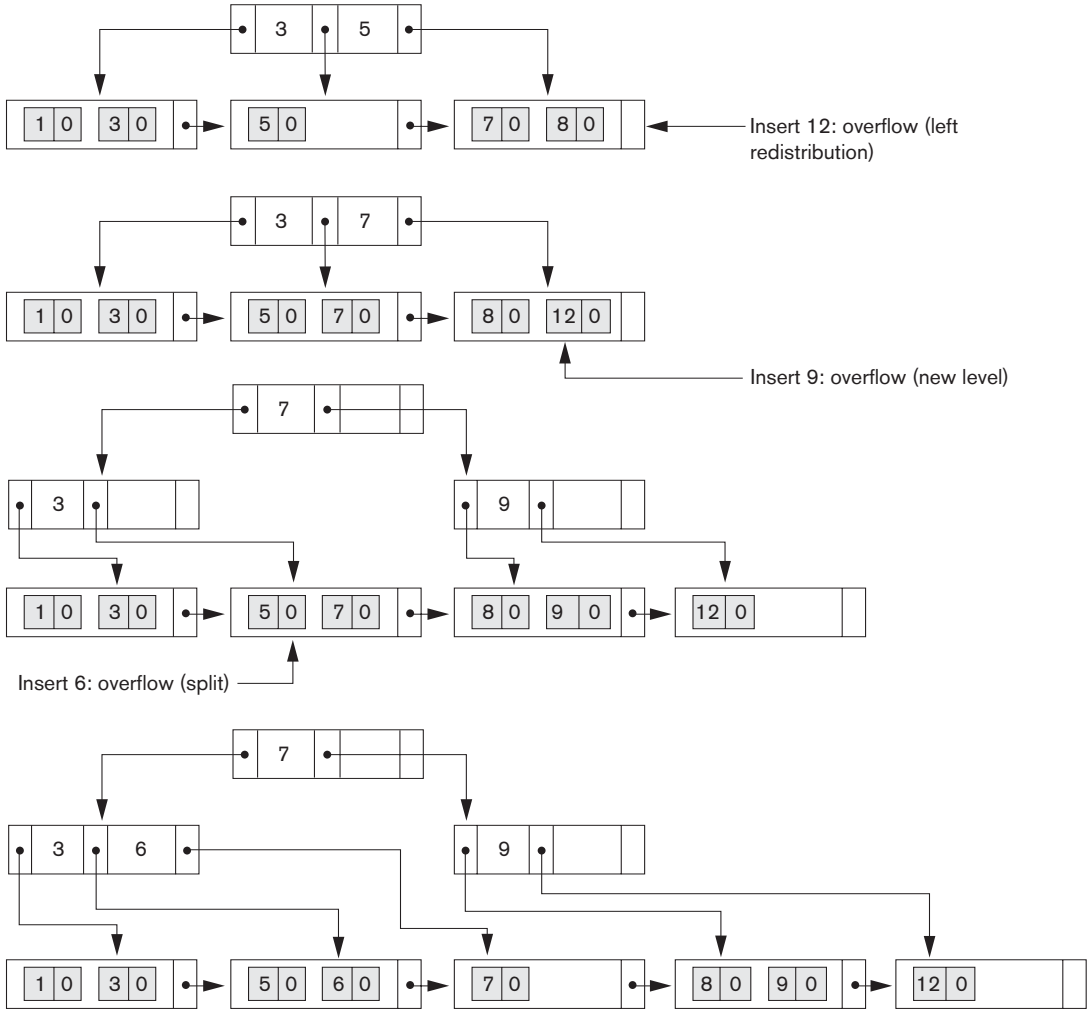


Figure 18.17
B^+-tree insertion with left redistribution.

part 8

Query Processing and Optimization, and Database Tuning

chapter 19

Algorithms for Query Processing and Optimization

In this chapter we discuss the techniques used internally by a DBMS to process, optimize, and execute high-level queries. A query expressed in a high-level query language such as SQL must first be scanned, parsed, and validated.[1] The **scanner** identifies the query tokens—such as SQL keywords, attribute names, and relation names—that appear in the text of the query, whereas the **parser** checks the query syntax to determine whether it is formulated according to the syntax rules (rules of grammar) of the query language. The query must also be **validated** by checking that all attribute and relation names are valid and semantically meaningful names in the schema of the particular database being queried. An internal representation of the query is then created, usually as a tree data structure called a **query tree**. It is also possible to represent the query using a graph data structure called a **query graph**. The DBMS must then devise an **execution strategy** or **query plan** for retrieving the results of the query from the database files. A query typically has many possible execution strategies, and the process of choosing a suitable one for processing a query is known as **query optimization**.

Figure 19.1 shows the different steps of processing a high-level query. The **query optimizer** module has the task of producing a good execution plan, and the **code generator** generates the code to execute that plan. The **runtime database processor** has the task of running (executing) the query code, whether in compiled or interpreted mode, to produce the query result. If a runtime error results, an error message is generated by the runtime database processor.

[1]We will not discuss the parsing and syntax-checking phase of query processing here; this material is discussed in compiler textbooks.

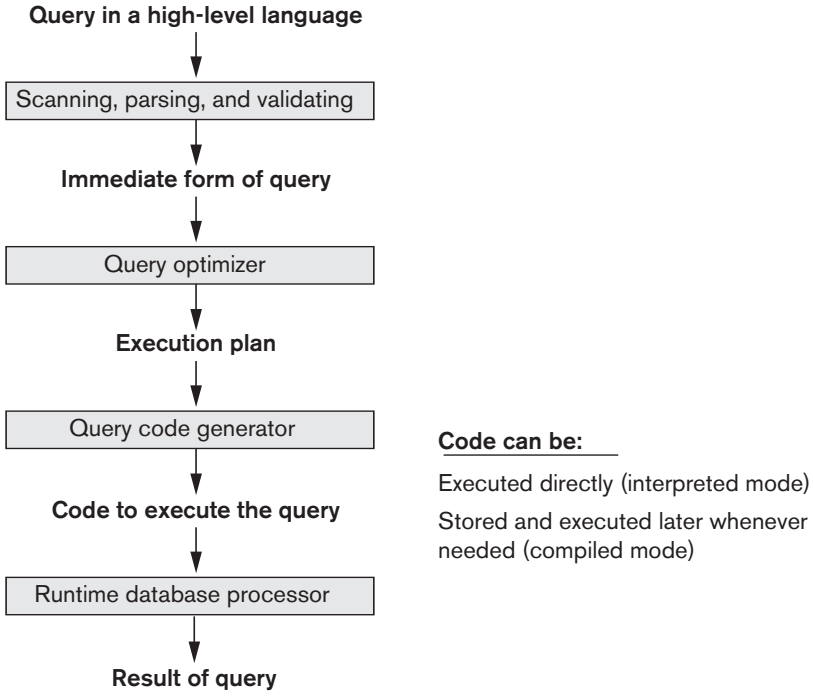


Figure 19.1
Typical steps when processing a high-level query.

The term *optimization* is actually a misnomer because in some cases the chosen execution plan is not the optimal (or absolute best) strategy—it is just a *reasonably efficient strategy* for executing the query. Finding the optimal strategy is usually too time-consuming—except for the simplest of queries. In addition, trying to find the optimal query execution strategy may require detailed information on how the files are implemented and even on the contents of the files—information that may not be fully available in the DBMS catalog. Hence, *planning of a good execution strategy* may be a more accurate description than *query optimization.*

For lower-level navigational database languages in legacy systems—such as the network DML or the hierarchical DL/1 (see Section 2.6)—the programmer must choose the query execution strategy while writing a database program. If a DBMS provides only a navigational language, there is *limited need or opportunity* for extensive query optimization by the DBMS; instead, the programmer is given the capability to choose the query execution strategy. On the other hand, a high-level query language—such as SQL for relational DBMSs (RDBMSs) or OQL (see Chapter 11) for object DBMSs (ODBMSs)—is more declarative in nature because it specifies what the intended results of the query are, rather than identifying the details of *how* the result should be obtained. Query optimization is thus necessary for queries that are specified in a high-level query language.

We will concentrate on describing query optimization in the *context of an RDBMS* because many of the techniques we describe have also been adapted for other types

of database management systems, such as ODBMSs.[2] A relational DBMS must systematically evaluate alternative query execution strategies and choose a reasonably efficient or near-optimal strategy. Each DBMS typically has a number of general database access algorithms that implement relational algebra operations such as SELECT or JOIN (see Chapter 6) or combinations of these operations. Only execution strategies that can be implemented by the DBMS access algorithms and that apply to the particular query, as well as to the *particular physical database design*, can be considered by the query optimization module.

This chapter starts with a general discussion of how SQL queries are typically translated into relational algebra queries and then optimized in Section 19.1. Then we discuss algorithms for implementing relational algebra operations in Sections 19.2 through 19.6. Following this, we give an overview of query optimization strategies. There are two main techniques that are employed during query optimization. The first technique is based on **heuristic rules** for ordering the operations in a query execution strategy. A heuristic is a rule that works well in most cases but is not guaranteed to work well in every case. The rules typically reorder the operations in a query tree. The second technique involves **systematically estimating** the cost of different execution strategies and choosing the execution plan with the lowest cost estimate. These techniques are usually combined in a query optimizer. We discuss heuristic optimization in Section 19.7 and cost estimation in Section 19.8. Then we provide a brief overview of the factors considered during query optimization in the Oracle commercial RDBMS in Section 19.9. Section 19.10 introduces the topic of semantic query optimization, in which known constraints are used as an aid to devising efficient query execution strategies.

The topics covered in this chapter require that the reader be familiar with the material presented in several earlier chapters. In particular, the chapters on SQL (Chapters 4 and 5), relational algebra (Chapter 6), and file structures and indexing (Chapters 17 and 18) are a prerequisite to this chapter. Also, it is important to note that the topic of query processing and optimization is vast, and we can only give an introduction to the basic principles and techniques in this chapter.

19.1 Translating SQL Queries into Relational Algebra

In practice, SQL is the query language that is used in most commercial RDBMSs. An SQL query is first translated into an equivalent extended relational algebra expression—represented as a query tree data structure—that is then optimized. Typically, SQL queries are decomposed into *query blocks,* which form the basic units that can be translated into the algebraic operators and optimized. A **query block** contains a single SELECT-FROM-WHERE expression, as well as GROUP BY and HAVING clauses if these are part of the block. Hence, nested queries within a query are identified as

[2]There are some query optimization problems and techniques that are pertinent only to ODBMSs. However, we do not discuss them here because we give only an introduction to query optimization.

separate query blocks. Because SQL includes aggregate operators—such as MAX, MIN, SUM, and COUNT—these operators must also be included in the extended algebra, as we discussed in Section 6.4.

Consider the following SQL query on the EMPLOYEE relation in Figure 3.5:

```
SELECT  Lname, Fname
FROM    EMPLOYEE
WHERE   Salary > ( SELECT  MAX (Salary)
                   FROM    EMPLOYEE
                   WHERE   Dno=5 );
```

This query retrieves the names of employees (from any department in the company) who earn a salary that is greater than the *highest salary in department 5*. The query includes a nested subquery and hence would be decomposed into two blocks. The inner block is:

```
( SELECT  MAX (Salary)
  FROM    EMPLOYEE
  WHERE   Dno=5 )
```

This retrieves the highest salary in department 5. The outer query block is:

```
SELECT  Lname, Fname
FROM    EMPLOYEE
WHERE   Salary > c
```

where c represents the result returned from the inner block. The inner block could be translated into the following extended relational algebra expression:

$$\Im_{\text{MAX Salary}}(\sigma_{\text{Dno=5}}(\text{EMPLOYEE}))$$

and the outer block into the expression:

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

The *query optimizer* would then choose an execution plan for each query block. Notice that in the above example, the inner block needs to be evaluated only once to produce the maximum salary of employees in department 5, which is then used—as the constant c—by the outer block. We called this a *nested query (without correlation with the outer query)* in Section 5.1.2. It is much harder to optimize the more complex *correlated nested queries* (see Section 5.1.3), where a tuple variable from the outer query block appears in the WHERE-clause of the inner query block.

19.2 Algorithms for External Sorting

Sorting is one of the primary algorithms used in query processing. For example, whenever an SQL query specifies an ORDER BY-clause, the query result must be sorted. Sorting is also a key component in sort-merge algorithms used for JOIN and other operations (such as UNION and INTERSECTION), and in duplicate elimination algorithms for the PROJECT operation (when an SQL query specifies the DISTINCT

option in the SELECT clause). We will discuss one of these algorithms in this section. Note that sorting of a particular file may be avoided if an appropriate index—such as a primary or clustering index (see Chapter 18)—exists on the desired file attribute to allow ordered access to the records of the file.

External sorting refers to sorting algorithms that are suitable for large files of records stored on disk that do not fit entirely in main memory, such as most database files.[3] The typical external sorting algorithm uses a **sort-merge strategy**, which starts by sorting small subfiles—called **runs**—of the main file and then merges the sorted runs, creating larger sorted subfiles that are merged in turn. The sort-merge algorithm, like other database algorithms, requires *buffer space* in main memory, where the actual sorting and merging of the runs is performed. The basic algorithm, outlined in Figure 19.2, consists of two phases: the sorting phase and the merging phase. The buffer space in main memory is part of the **DBMS cache**—an area in the computer's main memory that is controlled by the DBMS. The buffer space is divided into individual buffers, where each **buffer** is the same size in bytes as the size of one disk block. Thus, one buffer can hold the contents of exactly *one disk block*.

In the **sorting phase**, runs (portions or pieces) of the file that can fit in the available buffer space are read into main memory, sorted using an *internal* sorting algorithm, and written back to disk as temporary sorted subfiles (or runs). The size of each run and the **number of initial runs** (n_R) are dictated by the **number of file blocks** (b) and the **available buffer space** (n_B). For example, if the number of available main memory buffers $n_B = 5$ disk blocks and the size of the file $b = 1024$ disk blocks, then $n_R = \lceil (b/n_B) \rceil$ or 205 initial runs each of size 5 blocks (except the last run which will have only 4 blocks). Hence, after the sorting phase, 205 sorted runs (or 205 sorted subfiles of the original file) are stored as temporary subfiles on disk.

In the **merging phase**, the sorted runs are merged during one or more **merge passes**. Each merge pass can have one or more merge steps. The **degree of merging** (d_M) is the number of sorted subfiles that can be merged in each merge step. During each merge step, one buffer block is needed to hold one disk block from each of the sorted subfiles being merged, and one additional buffer is needed for containing one disk block of the merge result, which will produce a larger sorted file that is the result of merging several smaller sorted subfiles. Hence, d_M is the smaller of $(n_B - 1)$ and n_R, and the number of merge passes is $\lceil (\log_{dM}(n_R)) \rceil$. In our example where $n_B = 5$, $d_M = 4$ (four-way merging), so the 205 initial sorted runs would be merged 4 at a time in each step into 52 larger sorted subfiles at the end of the first merge pass. These 52 sorted files are then merged 4 at a time into 13 sorted files, which are then merged into 4 sorted files, and then finally into 1 fully sorted file, which means that *four passes* are needed.

[3] *Internal sorting algorithms* are suitable for sorting data structures, such as tables and lists, that can fit entirely in main memory. These algorithms are described in detail in data structures and algorithms books, and include techniques such as quick sort, heap sort, bubble sort, and many others. We do not discuss these here.

```
set     i ← 1;
        j ← b;              {size of the file in blocks}
        k ← n_B;            {size of buffer in blocks}
        m ← ⌈(j/k)⌉;

{Sorting Phase}
while (i ≤ m)
do {
        read next k blocks of the file into the buffer or if there are less than k blocks
           remaining, then read in the remaining blocks;
        sort the records in the buffer and write as a temporary subfile;
        i ← i + 1;
}

{Merging Phase: merge subfiles until only 1 remains}
set     i ← 1;
        p ← ⌈log_{k−1} m⌉   {p is the number of passes for the merging phase}
        j ← m;
while (i ≤ p)
do {
        n ← 1;
        q ← ⌈j/(k−1)⌉;  {number of subfiles to write in this pass}
        while (n ≤ q)
        do {
           read next k−1 subfiles or remaining subfiles (from previous pass)
               one block at a time;
           merge and write as new subfile one block at a time;
           n ← n + 1;
        }
        j ← q;
        i ← i + 1;
}
```

Figure 19.2
Outline of the sort-merge algorithm for external sorting.

The performance of the sort-merge algorithm can be measured in the number of disk block reads and writes (between the disk and main memory) before the sorting of the whole file is completed. The following formula approximates this cost:

$$(2 * b) + (2 * b * (\log_{dM} n_R))$$

The first term $(2 * b)$ represents the number of block accesses for the sorting phase, since each file block is accessed twice: once for reading into a main memory buffer and once for writing the sorted records back to disk into one of the sorted subfiles. The second term represents the number of block accesses for the merging phase. During each merge pass, a number of disk blocks approximately equal to the original file blocks b is read and written. Since the number of merge passes is $(\log_{dM} n_R)$, we get the total merge cost of $(2 * b * (\log_{dM} n_R))$.

The minimum number of main memory buffers needed is $n_B = 3$, which gives a d_M of 2 and an n_R of $\lceil (b/3) \rceil$. The minimum d_M of 2 gives the worst-case performance of the algorithm, which is:

$(2 * b) + (2 * (b * (\log_2 n_R)))$.

The following sections discuss the various algorithms for the operations of the relational algebra (see Chapter 6).

19.3 Algorithms for SELECT and JOIN Operations

19.3.1 Implementing the SELECT Operation

There are many algorithms for executing a SELECT operation, which is basically a search operation to locate the records in a disk file that satisfy a certain condition. Some of the search algorithms depend on the file having specific access paths, and they may apply only to certain types of selection conditions. We discuss some of the algorithms for implementing SELECT in this section. We will use the following operations, specified on the relational database in Figure 3.5, to illustrate our discussion:

OP1: $\sigma_{\text{Ssn} = \text{'123456789'}} (\text{EMPLOYEE})$

OP2: $\sigma_{\text{Dnumber} > 5} (\text{DEPARTMENT})$

OP3: $\sigma_{\text{Dno} = 5} (\text{EMPLOYEE})$

OP4: $\sigma_{\text{Dno} = 5 \text{ AND Salary} > 30000 \text{ AND Sex} = \text{'F'}} (\text{EMPLOYEE})$

OP5: $\sigma_{\text{Essn='123456789' AND Pno} = 10} (\text{WORKS\_ON})$

Search Methods for Simple Selection. A number of search algorithms are possible for selecting records from a file. These are also known as **file scans**, because they scan the records of a file to search for and retrieve records that satisfy a selection condition.[4] If the search algorithm involves the use of an index, the index search is called an **index scan**. The following search methods (S1 through S6) are examples of some of the search algorithms that can be used to implement a select operation:

- **S1—Linear search (brute force algorithm).** Retrieve *every record* in the file, and test whether its attribute values satisfy the selection condition. Since the records are grouped into disk blocks, each disk block is read into a main memory buffer, and then a search through the records within the disk block is conducted in main memory.

[4]A selection operation is sometimes called a **filter**, since it filters out the records in the file that do *not* satisfy the selection condition.

- **S2—Binary search.** If the selection condition involves an equality comparison on a key attribute on which the file is **ordered**, binary search—which is more efficient than linear search—can be used. An example is OP1 if Ssn is the ordering attribute for the EMPLOYEE file.[5]
- **S3a—Using a primary index.** If the selection condition involves an equality comparison on a **key attribute** with a primary index—for example, Ssn = '123456789' in OP1—use the primary index to retrieve the record. Note that this condition retrieves a single record (at most).
- **S3b—Using a hash key.** If the selection condition involves an equality comparison on a **key attribute** with a hash key—for example, Ssn = '123456789' in OP1—use the hash key to retrieve the record. Note that this condition retrieves a single record (at most).
- **S4—Using a primary index to retrieve multiple records.** If the comparison condition is >, >=, <, or <= on a key field with a primary index—for example, Dnumber > 5 in OP2—use the index to find the record satisfying the corresponding equality condition (Dnumber = 5), then retrieve all subsequent records in the (ordered) file. For the condition Dnumber < 5, retrieve all the preceding records.
- **S5—Using a clustering index to retrieve multiple records.** If the selection condition involves an equality comparison on a **nonkey attribute** with a clustering index—for example, Dno = 5 in OP3—use the index to retrieve all the records satisfying the condition.
- **S6—Using a secondary (B^+-tree) index on an equality comparison.** This search method can be used to retrieve a single record if the indexing field is a **key** (has unique values) or to retrieve multiple records if the indexing field is **not a key**. This can also be used for comparisons involving >, >=, <, or <=.

In Section 19.8, we discuss how to develop formulas that estimate the access cost of these search methods in terms of the number of block accesses and access time. Method S1 (**linear search**) applies to any file, but all the other methods depend on having the appropriate access path on the attribute used in the selection condition. Method S2 (**binary search**) requires the file to be sorted on the search attribute. The methods that use an index (S3a, S4, S5, and S6) are generally referred to as **index searches**, and they require the appropriate index to exist on the search attribute. Methods S4 and S6 can be used to retrieve records in a certain *range*—for example, 30000 <= Salary <= 35000. Queries involving such conditions are called **range queries**.

Search Methods for Complex Selection. If a condition of a SELECT operation is a **conjunctive condition**—that is, if it is made up of several simple conditions

[5]Generally, binary search is not used in database searches because ordered files are not used unless they also have a corresponding primary index.

connected with the AND logical connective such as OP4 above—the DBMS can use the following additional methods to implement the operation:

- **S7—Conjunctive selection using an individual index.** If an attribute involved in any **single simple condition** in the conjunctive select condition has an access path that permits the use of one of the methods S2 to S6, use that condition to retrieve the records and then check whether each retrieved record *satisfies the remaining simple conditions* in the conjunctive select condition.
- **S8—Conjunctive selection using a composite index.** If two or more attributes are involved in equality conditions in the conjunctive select condition and a composite index (or hash structure) exists on the combined fields—for example, if an index has been created on the composite key (Essn, Pno) of the WORKS_ON file for OP5—we can use the index directly.
- **S9—Conjunctive selection by intersection of record pointers.**[6] If secondary indexes (or other access paths) are available on more than one of the fields involved in simple conditions in the conjunctive select condition, and if the indexes include record pointers (rather than block pointers), then each index can be used to retrieve the **set of record pointers** that satisfy the individual condition. The **intersection** of these sets of record pointers gives the record pointers that satisfy the conjunctive select condition, which are then used to retrieve those records directly. If only some of the conditions have secondary indexes, each retrieved record is further tested to determine whether it satisfies the remaining conditions.[7] In general, method S9 assumes that each of the indexes is on a *nonkey field* of the file, because if one of the conditions is an equality condition on a key field, only one record will satisfy the whole condition.

Whenever a single condition specifies the selection—such as OP1, OP2, or OP3—the DBMS can only check whether or not an access path exists on the attribute involved in that condition. If an access path (such as index or hash key or sorted file) exists, the method corresponding to that access path is used; otherwise, the brute force, linear search approach of method S1 can be used. Query optimization for a SELECT operation is needed mostly for conjunctive select conditions whenever *more than one* of the attributes involved in the conditions have an access path. The optimizer should choose the access path that *retrieves the fewest records* in the most efficient way by estimating the different costs (see Section 19.8) and choosing the method with the least estimated cost.

Selectivity of a Condition. When the optimizer is choosing between multiple simple conditions in a conjunctive select condition, it typically considers the

[6]A record pointer uniquely identifies a record and provides the address of the record on disk; hence, it is also called the **record identifier** or **record id**.

[7]The technique can have many variations—for example, if the indexes are *logical indexes* that store primary key values instead of record pointers.

selectivity of each condition. The **selectivity (*sl*)** is defined as the ratio of the number of records (tuples) that satisfy the condition to the total number of records (tuples) in the file (relation), and thus is a number between zero and one. *Zero selectivity* means none of the records in the file satisfies the selection condition, and a selectivity of one means that all the records in the file satisfy the condition. In general, the selectivity will not be either of these two extremes, but will be a fraction that estimates the percentage of file records that will be retrieved.

Although exact selectivities of all conditions may not be available, **estimates of selectivities** are often kept in the DBMS catalog and are used by the optimizer. For example, for an equality condition on a key attribute of relation $r(R)$, $s = 1/|r(R)|$, where $|r(R)|$ is the number of tuples in relation $r(R)$. For an equality condition on a nonkey attribute with *i distinct values, s* can be estimated by $(|r(R)|/i)/|r(R)|$ or $1/i$, assuming that the records are evenly or **uniformly distributed** among the distinct values.[8] Under this assumption, $|r(R)|/i$ records will satisfy an equality condition on this attribute. In general, the number of records satisfying a selection condition with selectivity *sl* is estimated to be $|r(R)| * sl$. The smaller this estimate is, the higher the desirability of using that condition first to retrieve records. In certain cases, the actual distribution of records among the various distinct values of the attribute is kept by the DBMS in the form of a *histogram*, in order to get more accurate estimates of the number of records that satisfy a particular condition.

Disjunctive Selection Conditions. Compared to a conjunctive selection condition, a **disjunctive condition** (where simple conditions are connected by the OR logical connective rather than by AND) is much harder to process and optimize. For example, consider OP4′:

$$\text{OP4}'\text{: } \sigma_{\text{Dno=5 OR Salary > 30000 OR Sex='F'}}(\text{EMPLOYEE})$$

With such a condition, little optimization can be done, because the records satisfying the disjunctive condition are the *union* of the records satisfying the individual conditions. Hence, if any *one* of the conditions does not have an access path, we are compelled to use the brute force, linear search approach. Only if an access path exists on *every* simple condition in the disjunction can we optimize the selection by retrieving the records satisfying each condition—or their record ids—and then applying the *union* operation to eliminate duplicates.

A DBMS will have available many of the methods discussed above, and typically many additional methods. The query optimizer must choose the appropriate one for executing each SELECT operation in a query. This optimization uses formulas that estimate the costs for each available access method, as we will discuss in Section 19.8. The optimizer chooses the access method with the lowest estimated cost.

[8]In more sophisticated optimizers, histograms representing the distribution of the records among the different attribute values can be kept in the catalog.

19.3.2 Implementing the JOIN Operation

The JOIN operation is one of the most time-consuming operations in query processing. Many of the join operations encountered in queries are of the EQUIJOIN and NATURAL JOIN varieties, so we consider just these two here since we are only giving an overview of query processing and optimization. For the remainder of this chapter, the term **join** refers to an EQUIJOIN (or NATURAL JOIN).

There are many possible ways to implement a **two-way join**, which is a join on two files. Joins involving more than two files are called **multiway joins**. The number of possible ways to execute multiway joins grows very rapidly. In this section we discuss techniques for implementing *only two-way joins*. To illustrate our discussion, we refer to the relational schema in Figure 3.5 once more—specifically, to the EMPLOYEE, DEPARTMENT, and PROJECT relations. The algorithms we discuss next are for a join operation of the form:

$$R \bowtie_{A=B} S$$

where A and B are the **join attributes**, which should be domain-compatible attributes of R and S, respectively. The methods we discuss can be extended to more general forms of join. We illustrate four of the most common techniques for performing such a join, using the following sample operations:

$$\text{OP6: EMPLOYEE} \bowtie_{\text{Dno=Dnumber}} \text{DEPARTMENT}$$
$$\text{OP7: DEPARTMENT} \bowtie_{\text{Mgr\_ssn=Ssn}} \text{EMPLOYEE}$$

Methods for Implementing Joins.

- **J1—Nested-loop join (or nested-block join).** This is the default (brute force) algorithm, as it does not require any special access paths on either file in the join. For each record t in R (outer loop), retrieve every record s from S (inner loop) and test whether the two records satisfy the join condition $t[A] = s[B]$.[9]
- **J2—Single-loop join (using an access structure to retrieve the matching records).** If an index (or hash key) exists for one of the two join attributes—say, attribute B of file S—retrieve each record t in R (loop over file R), and then use the access structure (such as an index or a hash key) to retrieve directly all matching records s from S that satisfy $s[B] = t[A]$.
- **J3—Sort-merge join.** If the records of R and S are *physically sorted* (ordered) by value of the join attributes A and B, respectively, we can implement the join in the most efficient way possible. Both files are scanned concurrently in order of the join attributes, matching the records that have the same values for A and B. If the files are not sorted, they may be sorted first by using external sorting (see Section 19.2). In this method, pairs of file blocks are copied into memory buffers in order and the records of each file are scanned only once each for

[9]For disk files, it is obvious that the loops will be over disk blocks, so this technique has also been called *nested-block join*.

matching with the other file—unless both A and B are nonkey attributes, in which case the method needs to be modified slightly. A sketch of the sort-merge join algorithm is given in Figure 19.3(a). We use $R(i)$ to refer to the ith record in file R. A variation of the sort-merge join can be used when secondary indexes exist on both join attributes. The indexes provide the ability to access (scan) the records in order of the join attributes, but the records themselves are physically scattered all over the file blocks, so this method may be quite inefficient, as every record access may involve accessing a different disk block.

- **J4—Partition-hash join.** The records of files R and S are partitioned into smaller files. The partitioning of each file is done using the same hashing function h on the join attribute A of R (for partitioning file R) and B of S (for partitioning file S). First, a single pass through the file with fewer records (say, R) hashes its records to the various partitions of R; this is called the **partitioning phase**, since the records of R are partitioned into the hash buckets. In the simplest case, we assume that the smaller file can fit entirely in main memory after it is partitioned, so that the partitioned subfiles of R are all kept in main memory. The collection of records with the same value of $h(A)$ are placed in the same partition, which is a **hash bucket** in a hash table in main memory. In the second phase, called the **probing phase**, a single pass through the other file (S) then hashes each of its records using the same hash function $h(B)$ to *probe* the appropriate bucket, and that record is combined with all matching records from R in that bucket. This simplified description of partition-hash join assumes that the smaller of the two files *fits entirely into memory buckets* after the first phase. We will discuss the general case of partition-hash join that does not require this assumption below. In practice, techniques J1 to J4 are implemented by accessing *whole disk blocks* of a file, rather than individual records. Depending on the available number of buffers in memory, the number of blocks read in from the file can be adjusted.

How Buffer Space and Choice of Outer-Loop File Affect Performance of Nested-Loop Join. The buffer space available has an important effect on some of the join algorithms. First, let us consider the nested-loop approach (J1). Looking again at the operation OP6 above, assume that the number of buffers available in main memory for implementing the join is $n_B = 7$ blocks (buffers). Recall that we assume that each memory buffer is the same size as one disk block. For illustration, assume that the DEPARTMENT file consists of $r_D = 50$ records stored in $b_D = 10$ disk blocks and that the EMPLOYEE file consists of $r_E = 6000$ records stored in $b_E = 2000$ disk blocks. It is advantageous to read as many blocks as possible at a time into memory from the file whose records are used for the outer loop (that is, $n_B - 2$ blocks). The algorithm can then read one block at a time for the inner-loop file and use its records to **probe** (that is, search) the outer-loop blocks that are currently in main memory for matching records. This reduces the total number of block accesses. An extra buffer in main memory is needed to contain the resulting records after they are joined, and the contents of this result buffer can be appended to the **result file**—the disk file that will contain the join result—whenever it is filled. This result buffer block then is reused to hold additional join result records.

Figure 19.3
Implementing JOIN, PROJECT, UNION, INTERSECTION, and SET DIFFERENCE by using sort-merge, where R has n tuples and S has m tuples. (a) Implementing the operation $T \leftarrow R \bowtie_{A=B} S$. (b) Implementing the operation $T \leftarrow \pi_{<\text{attribute list}>}(R)$.

```
(a) sort the tuples in R on attribute A;                    (* assume R has n tuples (records) *)
    sort the tuples in S on attribute B;                    (* assume S has m tuples (records) *)
    set i ← 1, j ← 1;
    while (i ≤ n) and (j ≤ m)
    do {  if R(i) [A] > S(j) [B]
             then  set j ← j + 1
          elseif R(i) [A] < S(j) [B]
             then  set i ← i + 1
          else  {   (* R(i) [A] = S(j) [B], so we output a matched tuple *)
                    output the combined tuple <R(i), S(j)> to T;

                    (* output other tuples that match R(i), if any *)
                    set l ← j + 1;
                    while (l ≤ m) and (R(i) [A] = S(l) [B])
                    do {  output the combined tuple <R(i), S(l)> to T;
                          set l ← l + 1
                    }

                 (* output other tuples that match S(j), if any *)
                 set k ← i + 1;
                 while (k ≤ n) and (R(k) [A] = S(j) [B])
                 do {  output the combined tuple <R(k), S(j)> to T;
                       set k ← k + 1
                 }
                 set i ← k, j ← l
          }
    }

(b) create a tuple t[<attribute list>] in T′ for each tuple t in R;
       (* T′ contains the projection results before duplicate elimination *)
    if <attribute list> includes a key of R
       then T ← T′
    else  {  sort the tuples in T′;
             set i ← 1, j ← 2;
             while i ≤ n
             do {  output the tuple T′ [i] to T;
                   while T′ [i] = T′ [j] and j ≤ n do j ← j + 1;        (* eliminate duplicates *)
                   i ← j; j ← i + 1
             }
}
(* T contains the projection result after duplicate elimination *)
```

(continues)

Figure 19.3 (continued)
Implementing JOIN, PROJECT, UNION, INTERSECTION, and SET DIFFERENCE by using sort-merge, where R has n tuples and S has m tuples. (c) Implementing the operation $T \leftarrow R \cup S$. (d) Implementing the operation $T \leftarrow R \cap S$. (e) Implementing the operation $T \leftarrow R - S$.

(c)
```
sort the tuples in R and S using the same unique sort attributes;
set i ← 1, j ← 1;
while (i ≤ n) and (j ≤ m)
do {  if R(i) > S(j)
         then  {  output S(j) to T;
                  set j ← j + 1
               }
      elseif R(i) < S(j)
         then  {  output R(i) to T;
                  set i ← i + 1
               }
      else set  j ← j + 1                    (* R(i)=S(j), so we skip one of the duplicate tuples *)
}
if (i ≤ n) then add tuples R(i) to R(n) to T;
if (j ≤ m) then add tuples S(j) to S(m) to T;
```

(d)
```
sort the tuples in R and S using the same unique sort attributes;
set i ← 1, j ← 1;
while (i ≤ n) and (j ≤ m)
do {  if R(i) > S(j)
         then  set j ← j + 1
      elseif R(i) < S(j)
         then  set i ← i + 1
      else  {  output R(j) to T;             (* R(i)=S(j), so we output the tuple *)
               set i ← i + 1, j ← j + 1
      }
}
```

(e)
```
sort the tuples in R and S using the same unique sort attributes;
set i ← 1, j ← 1;
while (i ≤ n) and (j ≤ m)
do {  if R(i) > S(j)
         then  set j ← j + 1
      elseif R(i) < S(j)
         then  {  output R(i) to T;          (* R(i) has no matching S(j), so output R(i) *)
                  set i ← i + 1
               }
      else  set i ← i + 1, j ← j + 1
}
if (i ≤ n) then add tuples R(i) to R(n) to T;
```

In the nested-loop join, it makes a difference which file is chosen for the outer loop and which for the inner loop. If EMPLOYEE is used for the outer loop, each block of EMPLOYEE is read once, and the entire DEPARTMENT file (each of its blocks) is read once for *each time* we read in $(n_B - 2)$ blocks of the EMPLOYEE file. We get the following formulas for the number of disk blocks that are read from disk to main memory:

Total number of blocks accessed (read) for outer-loop file = b_E

Number of times $(n_B - 2)$ blocks of outer file are loaded into main memory $= \lceil b_E/(n_B - 2) \rceil$

Total number of blocks accessed (read) for inner-loop file = $b_D * \lceil b_E/(n_B - 2) \rceil$

Hence, we get the following total number of block read accesses:

$$b_E + (\lceil b_E/(n_B - 2) \rceil * b_D) = 2000 + (\lceil (2000/5) \rceil * 10) = 6000 \text{ block accesses}$$

On the other hand, if we use the DEPARTMENT records in the outer loop, by symmetry we get the following total number of block accesses:

$$b_D + (\lceil b_D/(n_B - 2) \rceil * b_E) = 10 + (\lceil (10/5) \rceil * 2000) = 4010 \text{ block accesses}$$

The join algorithm uses a buffer to hold the joined records of the result file. Once the buffer is filled, it is written to disk and its contents are appended to the result file, and then refilled with join result records.[10]

If the result file of the join operation has b_{RES} disk blocks, each block is written once to disk, so an additional b_{RES} block accesses (writes) should be added to the preceding formulas in order to estimate the total cost of the join operation. The same holds for the formulas developed later for other join algorithms. As this example shows, it is advantageous to use the file *with fewer blocks* as the outer-loop file in the nested-loop join.

How the Join Selection Factor Affects Join Performance. Another factor that affects the performance of a join, particularly the single-loop method J2, is the fraction of records in one file that will be joined with records in the other file. We call this the **join selection factor**[11] of a file with respect to an equijoin condition with another file. This factor depends on the particular equijoin condition between the two files. To illustrate this, consider the operation OP7, which joins each DEPARTMENT record with the EMPLOYEE record for the manager of that department. Here, each DEPARTMENT record (there are 50 such records in our example) will be joined with a *single* EMPLOYEE record, but many EMPLOYEE records (the 5,950 of them that do not manage a department) will not be joined with any record from DEPARTMENT.

Suppose that secondary indexes exist on both the attributes Ssn of EMPLOYEE and Mgr_ssn of DEPARTMENT, with the number of index levels $x_{\text{Ssn}} = 4$ and $x_{\text{Mgr\_ssn}} = 2$,

[10] If we reserve two buffers for the result file, double buffering can be used to speed the algorithm (see Section 17.3).

[11] This is different from the *join selectivity*, which we will discuss in Section 19.8.

respectively. We have two options for implementing method J2. The first retrieves each EMPLOYEE record and then uses the index on Mgr_ssn of DEPARTMENT to find a matching DEPARTMENT record. In this case, no matching record will be found for employees who do not manage a department. The number of block accesses for this case is approximately:

$$b_E + (r_E * (x_{\text{Mgr\_ssn}} + 1)) = 2000 + (6000 * 3) = 20{,}000 \text{ block accesses}$$

The second option retrieves each DEPARTMENT record and then uses the index on Ssn of EMPLOYEE to find a matching manager EMPLOYEE record. In this case, every DEPARTMENT record will have one matching EMPLOYEE record. The number of block accesses for this case is approximately:

$$b_D + (r_D * (x_{\text{Ssn}} + 1)) = 10 + (50 * 5) = 260 \text{ block accesses}$$

The second option is more efficient because the join selection factor of DEPARTMENT *with respect to the join condition* Ssn = Mgr_ssn is 1 (every record in DEPARTMENT will be joined), whereas the join selection factor of EMPLOYEE with respect to the same join condition is (50/6000), or 0.008 (only 0.8 percent of the records in EMPLOYEE will be joined). For method J2, either the smaller file or the file that has a match for every record (that is, the file with the high join selection factor) should be used in the (single) join loop. It is also possible to create an index specifically for performing the join operation if one does not already exist.

The sort-merge join J3 is quite efficient if both files are already sorted by their join attribute. Only a single pass is made through each file. Hence, the number of blocks accessed is equal to the sum of the numbers of blocks in both files. For this method, both OP6 and OP7 would need $b_E + b_D = 2000 + 10 = 2010$ block accesses. However, both files are required to be ordered by the join attributes; if one or both are not, a sorted copy of each file must be created specifically for performing the join operation. If we roughly estimate the cost of sorting an external file by $(b \log_2 b)$ block accesses, and if both files need to be sorted, the total cost of a sort-merge join can be estimated by $(b_E + b_D + b_E \log_2 b_E + b_D \log_2 b_D)$.[12]

General Case for Partition-Hash Join. The hash-join method J4 is also quite efficient. In this case only a single pass is made through each file, whether or not the files are ordered. If the hash table for the smaller of the two files can be kept entirely in main memory after hashing (partitioning) on its join attribute, the implementation is straightforward. If, however, the partitions of both files must be stored on disk, the method becomes more complex, and a number of variations to improve the efficiency have been proposed. We discuss two techniques: the general case of *partition-hash join* and a variation called *hybrid hash-join algorithm*, which has been shown to be quite efficient.

In the general case of **partition-hash join**, each file is first partitioned into M partitions using the same **partitioning hash function** on the join attributes. Then, each

[12]We can use the more accurate formulas from Section 19.2 if we know the number of available buffers for sorting.

pair of corresponding partitions is joined. For example, suppose we are joining relations R and S on the join attributes $R.A$ and $S.B$:

$$R \bowtie_{A=B} S$$

In the **partitioning phase**, R is partitioned into the M partitions $R_1, R_2, ..., R_M$, and S into the M partitions $S_1, S_2, ..., S_M$. The property of each pair of corresponding partitions R_i, S_i with respect to the join operation is that records in R_i *only need to be joined* with records in S_i, and vice versa. This property is ensured by using the *same hash function* to partition both files on their join attributes—attribute A for R and attribute B for S. The minimum number of in-memory buffers needed for the **partitioning phase** is $M + 1$. Each of the files R and S are partitioned separately. During partitioning of a file, M in-memory buffers are allocated to store the records that hash to each partition, and one additional buffer is needed to hold one block at a time of the input file being partitioned. Whenever the in-memory buffer for a partition gets filled, its contents are appended to a **disk subfile** that stores the partition. The partitioning phase has *two iterations.* After the first iteration, the first file R is partitioned into the subfiles $R_1, R_2, ..., R_M$, where all the records that hashed to the same buffer are in the same partition. After the second iteration, the second file S is similarly partitioned.

In the second phase, called the **joining** or **probing phase**, M *iterations* are needed. During iteration i, two corresponding partitions R_i and S_i are joined. The minimum number of buffers needed for iteration i is the number of blocks in the smaller of the two partitions, say R_i, plus two additional buffers. If we use a nested-loop join during iteration i, the records from the smaller of the two partitions R_i are copied into memory buffers; then all blocks from the other partition S_i are read—one at a time—and each record is used to **probe** (that is, search) partition R_i for matching record(s). Any matching records are joined and written into the result file. To improve the efficiency of in-memory probing, it is common to use an *in-memory hash table* for storing the records in partition R_i by using a *different* hash function from the partitioning hash function.[13]

We can approximate the cost of this partition hash-join as $3 * (b_R + b_S) + b_{RES}$ for our example, since each record is read once and written back to disk once during the partitioning phase. During the joining (probing) phase, each record is read a second time to perform the join. The *main difficulty* of this algorithm is to ensure that the partitioning hash function is **uniform**—that is, the partition sizes are nearly equal in size. If the partitioning function is **skewed** (nonuniform), then some partitions may be too large to fit in the available memory space for the second joining phase.

Notice that if the available in-memory buffer space $n_B > (b_R + 2)$, where b_R is the number of blocks for the *smaller* of the two files being joined, say R, then there is no reason to do partitioning since in this case the join can be performed entirely in memory using some variation of the nested-loop join based on hashing and probing.

[13]If the hash function used for partitioning is used again, all records in a partition will hash to the same bucket again.

For illustration, assume we are performing the join operation OP6, repeated below:

OP6: EMPLOYEE $\bowtie_{\text{Dno=Dnumber}}$ DEPARTMENT

In this example, the smaller file is the DEPARTMENT file; hence, if the number of available memory buffers $n_B > (b_D + 2)$, the whole DEPARTMENT file can be read into main memory and organized into a hash table on the join attribute. Each EMPLOYEE block is then read into a buffer, and each EMPLOYEE record in the buffer is hashed on its join attribute and is used to *probe* the corresponding in-memory bucket in the DEPARTMENT hash table. If a matching record is found, the records are joined, and the result record(s) are written to the result buffer and eventually to the result file on disk. The cost in terms of block accesses is hence $(b_D + b_E)$, plus b_{RES}—the cost of writing the result file.

Hybrid Hash-Join. The **hybrid hash-join algorithm** is a variation of partition hash-join, where the *joining* phase for *one of the partitions* is included in the *partitioning* phase. To illustrate this, let us assume that the size of a memory buffer is one disk block; that n_B such buffers are *available*; and that the partitioning hash function used is $h(K) = K \bmod M$, so that M partitions are being created, where $M < n_B$. For illustration, assume we are performing the join operation OP6. In the *first pass* of the partitioning phase, when the hybrid hash-join algorithm is partitioning the smaller of the two files (DEPARTMENT in OP6), the algorithm divides the buffer space among the M partitions such that all the blocks of the *first partition* of DEPARTMENT completely reside in main memory. For each of the other partitions, only a single in-memory buffer—whose size is one disk block—is allocated; the remainder of the partition is written to disk as in the regular partition-hash join. Hence, at the end of the *first pass of the partitioning phase,* the first partition of DEPARTMENT resides wholly in main memory, whereas each of the other partitions of DEPARTMENT resides in a disk subfile.

For the second pass of the partitioning phase, the records of the second file being joined—the larger file, EMPLOYEE in OP6—are being partitioned. If a record hashes to the *first partition,* it is joined with the matching record in DEPARTMENT and the joined records are written to the result buffer (and eventually to disk). If an EMPLOYEE record hashes to a partition other than the first, it is partitioned normally and stored to disk. Hence, at the end of the second pass of the partitioning phase, all records that hash to the first partition have been joined. At this point, there are $M - 1$ pairs of partitions on disk. Therefore, during the second **joining** or **probing** phase, $M - 1$ *iterations* are needed instead of M. The goal is to join as many records during the partitioning phase so as to save the cost of storing those records on disk and then rereading them a second time during the joining phase.

19.4 Algorithms for PROJECT and Set Operations

A PROJECT operation $\pi_{\text{<attribute list>}}(R)$ is straightforward to implement if <attribute list> includes a key of relation R, because in this case the result of the operation will

have the same number of tuples as R, but with only the values for the attributes in <attribute list> in each tuple. If <attribute list> does not include a key of R, *duplicate tuples must be eliminated.* This can be done by sorting the result of the operation and then eliminating duplicate tuples, which appear consecutively after sorting. A sketch of the algorithm is given in Figure 19.3(b). Hashing can also be used to eliminate duplicates: as each record is hashed and inserted into a bucket of the hash file in memory, it is checked against those records already in the bucket; if it is a duplicate, it is not inserted in the bucket. It is useful to recall here that in SQL queries, the default is not to eliminate duplicates from the query result; duplicates are eliminated from the query result only if the keyword DISTINCT is included.

Set operations—UNION, INTERSECTION, SET DIFFERENCE, and CARTESIAN PRODUCT—are sometimes expensive to implement. In particular, the CARTESIAN PRODUCT operation $R \times S$ is quite expensive because its result includes a record for each combination of records from R and S. Also, each record in the result includes all attributes of R and S. If R has n records and j attributes, and S has m records and k attributes, the result relation for $R \times S$ will have $n * m$ records and each record will have $j + k$ attributes. Hence, it is important to avoid the CARTESIAN PRODUCT operation and to substitute other operations such as join during query optimization (see Section 19.7).

The other three set operations—UNION, INTERSECTION, and SET DIFFERENCE[14]—apply only to **type-compatible** (or union-compatible) relations, which have the same number of attributes and the same attribute domains. The customary way to implement these operations is to use variations of the **sort-merge technique:** the two relations are sorted on the same attributes, and, after sorting, a single scan through each relation is sufficient to produce the result. For example, we can implement the UNION operation, $R \cup S$, by scanning and merging both sorted files concurrently, and whenever the same tuple exists in both relations, only one is kept in the merged result. For the INTERSECTION operation, $R \cap S$, we keep in the merged result only those tuples that appear in *both sorted relations.* Figure 19.3(c) to (e) sketches the implementation of these operations by sorting and merging. Some of the details are not included in these algorithms.

Hashing can also be used to implement UNION, INTERSECTION, and SET DIFFERENCE. One table is first scanned and then partitioned into an in-memory hash table with buckets, and the records in the other table are then scanned one at a time and used to probe the appropriate partition. For example, to implement $R \cup S$, first hash (partition) the records of R; then, hash (probe) the records of S, but do not insert duplicate records in the buckets. To implement $R \cap S$, first partition the records of R to the hash file. Then, while hashing each record of S, probe to check if an identical record from R is found in the bucket, and if so add the record to the result file. To implement $R - S$, first hash the records of R to the hash file buckets. While hashing (probing) each record of S, if an identical record is found in the bucket, remove that record from the bucket.

[14]SET DIFFERENCE is called EXCEPT in SQL.

In SQL, there are two variations of these set operations. The operations UNION, INTERSECTION, and EXCEPT (the SQL keyword for the SET DIFFERENCE operation) apply to traditional sets, where no duplicate records exist in the result. The operations UNION ALL, INTERSECTION ALL, and EXCEPT ALL apply to multisets (or bags), and duplicates are fully considered. Variations of the above algorithms can be used for the multiset operations in SQL. We leave these as an exercise for the reader.

19.5 Implementing Aggregate Operations and OUTER JOINs

19.5.1 Implementing Aggregate Operations

The aggregate operators (MIN, MAX, COUNT, AVERAGE, SUM), when applied to an entire table, can be computed by a table scan or by using an appropriate index, if available. For example, consider the following SQL query:

```
SELECT  MAX(Salary)
FROM    EMPLOYEE;
```

If an (ascending) B^+-tree index on Salary exists for the EMPLOYEE relation, then the optimizer can decide on using the Salary index to search for the largest Salary value in the index by following the *rightmost* pointer in each index node from the root to the rightmost leaf. That node would include the largest Salary value as its *last* entry. In most cases, this would be more efficient than a full table scan of EMPLOYEE, since no actual records need to be retrieved. The MIN function can be handled in a similar manner, except that the *leftmost* pointer in the index is followed from the root to leftmost leaf. That node would include the smallest Salary value as its *first* entry.

The index could also be used for the AVERAGE and SUM aggregate functions, but only if it is a **dense index**—that is, if there is an index entry for every record in the main file. In this case, the associated computation would be applied to the values in the index. For a **nondense index**, the actual number of records associated with each index value must be used for a correct computation. This can be done if the *number of records associated with each value* in the index is stored in each index entry. For the COUNT aggregate function, the number of values can be also computed from the index in a similar manner. If a COUNT(*) function is applied to a whole relation, the number of records currently in each relation are typically stored in the catalog, and so the result can be retrieved directly from the catalog.

When a GROUP BY clause is used in a query, the aggregate operator must be applied separately to each group of tuples as partitioned by the grouping attribute. Hence, the table must first be partitioned into subsets of tuples, where each partition (group) has the same value for the grouping attributes. In this case, the computation is more complex. Consider the following query:

```
SELECT  Dno, AVG(Salary)
FROM    EMPLOYEE
GROUP BY Dno;
```

The usual technique for such queries is to first use either **sorting** or **hashing** on the grouping attributes to partition the file into the appropriate groups. Then the algorithm computes the aggregate function for the tuples in each group, which have the same grouping attribute(s) value. In the sample query, the set of EMPLOYEE tuples for each department number would be grouped together in a partition and the average salary computed for each group.

Notice that if a **clustering index** (see Chapter 18) exists on the grouping attribute(s), then the records are *already partitioned* (grouped) into the appropriate subsets. In this case, it is only necessary to apply the computation to each group.

19.5.2 Implementing OUTER JOINs

In Section 6.4, the *outer join operation* was discussed, with its three variations: left outer join, right outer join, and full outer join. We also discussed in Chapter 5 how these operations can be specified in SQL. The following is an example of a left outer join operation in SQL:

```
SELECT  Lname, Fname, Dname
FROM    (EMPLOYEE LEFT OUTER JOIN DEPARTMENT ON Dno=Dnumber);
```

The result of this query is a table of employee names and their associated departments. It is similar to a regular (inner) join result, with the exception that if an EMPLOYEE tuple (a tuple in the *left* relation) *does not have an associated department,* the employee's name will still appear in the resulting table, but the department name would be NULL for such tuples in the query result.

Outer join can be computed by modifying one of the join algorithms, such as nested-loop join or single-loop join. For example, to compute a *left* outer join, we use the left relation as the outer loop or single-loop because every tuple in the left relation must appear in the result. If there are matching tuples in the other relation, the joined tuples are produced and saved in the result. However, if no matching tuple is found, the tuple is still included in the result but is padded with NULL value(s). The sort-merge and hash-join algorithms can also be extended to compute outer joins.

Theoretically, outer join can also be computed by executing a combination of relational algebra operators. For example, the left outer join operation shown above is equivalent to the following sequence of relational operations:

1. Compute the (inner) JOIN of the EMPLOYEE and DEPARTMENT tables.
   $\text{TEMP1} \leftarrow \pi_{\text{Lname, Fname, Dname}} (\text{EMPLOYEE} \bowtie \text{Dno=Dnumber DEPARTMENT})$
2. Find the EMPLOYEE tuples that do not appear in the (inner) JOIN result.
   $\text{TEMP2} \leftarrow \pi_{\text{Lname, Fname}} (\text{EMPLOYEE}) - \pi_{\text{Lname, Fname}} (\text{TEMP1})$
3. Pad each tuple in TEMP2 with a NULL Dname field.
   $\text{TEMP2} \leftarrow \text{TEMP2} \times \text{NULL}$

4. Apply the UNION operation to TEMP1, TEMP2 to produce the LEFT OUTER JOIN result.

   RESULT ← TEMP1 ∪ TEMP2

The cost of the outer join as computed above would be the sum of the costs of the associated steps (inner join, projections, set difference, and union). However, note that step 3 can be done as the temporary relation is being constructed in step 2; that is, we can simply pad each resulting tuple with a NULL. In addition, in step 4, we know that the two operands of the union are disjoint (no common tuples), so there is no need for duplicate elimination.

19.6 Combining Operations Using Pipelining

A query specified in SQL will typically be translated into a relational algebra expression that is *a sequence of relational operations.* If we execute a single operation at a time, we must generate temporary files on disk to hold the results of these temporary operations, creating excessive overhead. Generating and storing large temporary files on disk is time-consuming and can be unnecessary in many cases, since these files will immediately be used as input to the next operation. To reduce the number of temporary files, it is common to generate query execution code that corresponds to algorithms for combinations of operations in a query.

For example, rather than being implemented separately, a JOIN can be combined with two SELECT operations on the input files and a final PROJECT operation on the resulting file; all this is implemented by one algorithm with two input files and a single output file. Rather than creating four temporary files, we apply the algorithm directly and get just one result file. In Section 19.7.2, we discuss how heuristic relational algebra optimization can group operations together for execution. This is called **pipelining** or **stream-based processing**.

It is common to create the query execution code dynamically to implement multiple operations. The generated code for producing the query combines several algorithms that correspond to individual operations. As the result tuples from one operation are produced, they are provided as input for subsequent operations. For example, if a join operation follows two select operations on base relations, the tuples resulting from each select are provided as input for the join algorithm in a **stream** or **pipeline** as they are produced.

19.7 Using Heuristics in Query Optimization

In this section we discuss optimization techniques that apply heuristic rules to modify the internal representation of a query—which is usually in the form of a query tree or a query graph data structure—to improve its expected performance. The scanner and parser of an SQL query first generate a data structure that corresponds to an *initial query representation,* which is then optimized according to heuristic rules. This leads to an *optimized query representation*, which corresponds to the query execution strategy. Following that, a query execution plan is generated

to execute groups of operations based on the access paths available on the files involved in the query.

One of the main **heuristic rules** is to apply SELECT and PROJECT operations *before* applying the JOIN or other binary operations, because the size of the file resulting from a binary operation—such as JOIN—is usually a multiplicative function of the sizes of the input files. The SELECT and PROJECT operations reduce the size of a file and hence should be applied *before* a join or other binary operation.

In Section 19.7.1 we reiterate the query tree and query graph notations that we introduced earlier in the context of relational algebra and calculus in Sections 6.3.5 and 6.6.5, respectively. These can be used as the basis for the data structures that are used for internal representation of queries. A *query tree* is used to represent a *relational algebra* or extended relational algebra expression, whereas a *query graph* is used to represent a *relational calculus expression*. Then in Section 19.7.2 we show how heuristic optimization rules are applied to convert an initial query tree into an **equivalent query tree**, which represents a different relational algebra expression that is more efficient to execute but gives the same result as the original tree. We also discuss the equivalence of various relational algebra expressions. Finally, Section 19.7.3 discusses the generation of query execution plans.

19.7.1 Notation for Query Trees and Query Graphs

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

Figure 19.4a shows a query tree (the same as shown in Figure 6.9) for query Q2 in Chapters 4 to 6: For every project located in 'Stafford', retrieve the project number, the controlling department number, and the department manager's last name, address, and birthdate. This query is specified on the COMPANY relational schema in Figure 3.5 and corresponds to the following relational algebra expression:

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

This corresponds to the following SQL query:

```
Q2: SELECT  P.Pnumber, P.Dnum, E.Lname, E.Address, E.Bdate
    FROM    PROJECT AS P, DEPARTMENT AS D, EMPLOYEE AS E
    WHERE   P.Dnum=D.Dnumber AND D.Mgr_ssn=E.Ssn AND
            P.Plocation= 'Stafford';
```

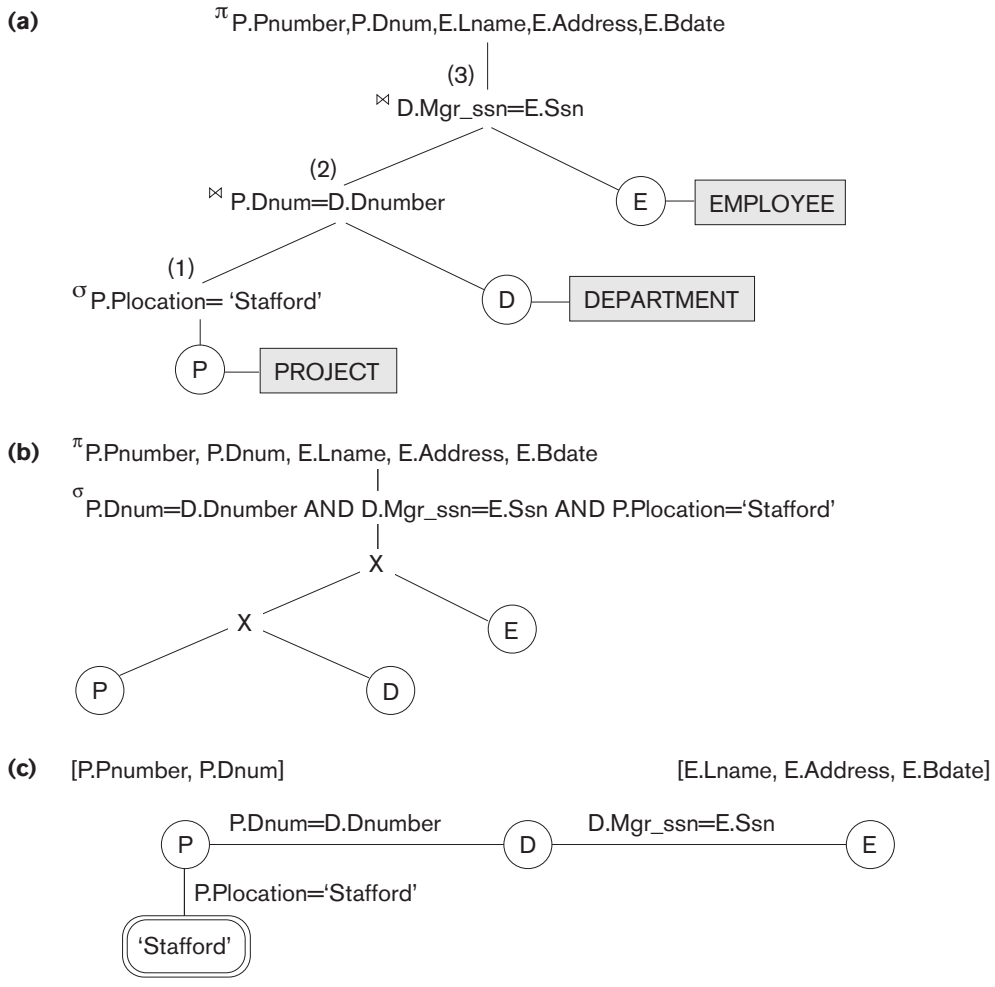


Figure 19.4
Two query trees for the query Q2. (a) Query tree corresponding to the relational algebra expression for Q2. (b) Initial (canonical) query tree for SQL query Q2. (c) Query graph for Q2.

In Figure 19.4a, the leaf nodes P, D, and E represent the three relations PROJECT, DEPARTMENT, and EMPLOYEE, respectively, and the internal tree nodes represent the *relational algebra operations* of the expression. When this query tree is executed, the node marked (1) in Figure 19.4a must begin execution before node (2) because some resulting tuples of operation (1) must be available before we can begin executing operation (2). Similarly, node (2) must begin executing and producing results before node (3) can start execution, and so on.

As we can see, the query tree represents a specific order of operations for executing a query. A more neutral data structure for representation of a query is the **query graph** notation. Figure 19.4c (the same as shown in Figure 6.13) shows the query

graph for query Q2. Relations in the query are represented by **relation nodes**, which are displayed as single circles. Constant values, typically from the query selection conditions, are represented by **constant nodes**, which are displayed as double circles or ovals. Selection and join conditions are represented by the graph **edges**, as shown in Figure 19.4c. Finally, the attributes to be retrieved from each relation are displayed in square brackets above each relation.

The query graph representation does not indicate an order on which operations to perform first. There is only a single graph corresponding to each query.[15] Although some optimization techniques were based on query graphs, it is now generally accepted that query trees are preferable because, in practice, the query optimizer needs to show the order of operations for query execution, which is not possible in query graphs.

19.7.2 Heuristic Optimization of Query Trees

In general, many different relational algebra expressions—and hence many different query trees—can be **equivalent**; that is, they can represent the *same query*.[16]

The query parser will typically generate a standard **initial query tree** to correspond to an SQL query, without doing any optimization. For example, for a SELECT-PROJECT-JOIN query, such as Q2, the initial tree is shown in Figure 19.4(b). The CARTESIAN PRODUCT of the relations specified in the FROM clause is first applied; then the selection and join conditions of the WHERE clause are applied, followed by the projection on the SELECT clause attributes. Such a canonical query tree represents a relational algebra expression that is *very inefficient if executed directly,* because of the CARTESIAN PRODUCT ($\times$) operations. For example, if the PROJECT, DEPARTMENT, and EMPLOYEE relations had record sizes of 100, 50, and 150 bytes and contained 100, 20, and 5,000 tuples, respectively, the result of the CARTESIAN PRODUCT would contain 10 million tuples of record size 300 bytes each. However, the initial query tree in Figure 19.4(b) is in a simple standard form that can be easily created from the SQL query. It will never be executed. The heuristic query optimizer will transform this initial query tree into an equivalent **final query tree** that is efficient to execute.

The optimizer must include rules for *equivalence among relational algebra expressions* that can be applied to transform the initial tree into the final, optimized query tree. First we discuss informally how a query tree is transformed by using heuristics, and then we discuss general transformation rules and show how they can be used in an algebraic heuristic optimizer.

Example of Transforming a Query. Consider the following query Q on the database in Figure 3.5: *Find the last names of employees born after 1957 who work on a project named 'Aquarius'*. This query can be specified in SQL as follows:

---

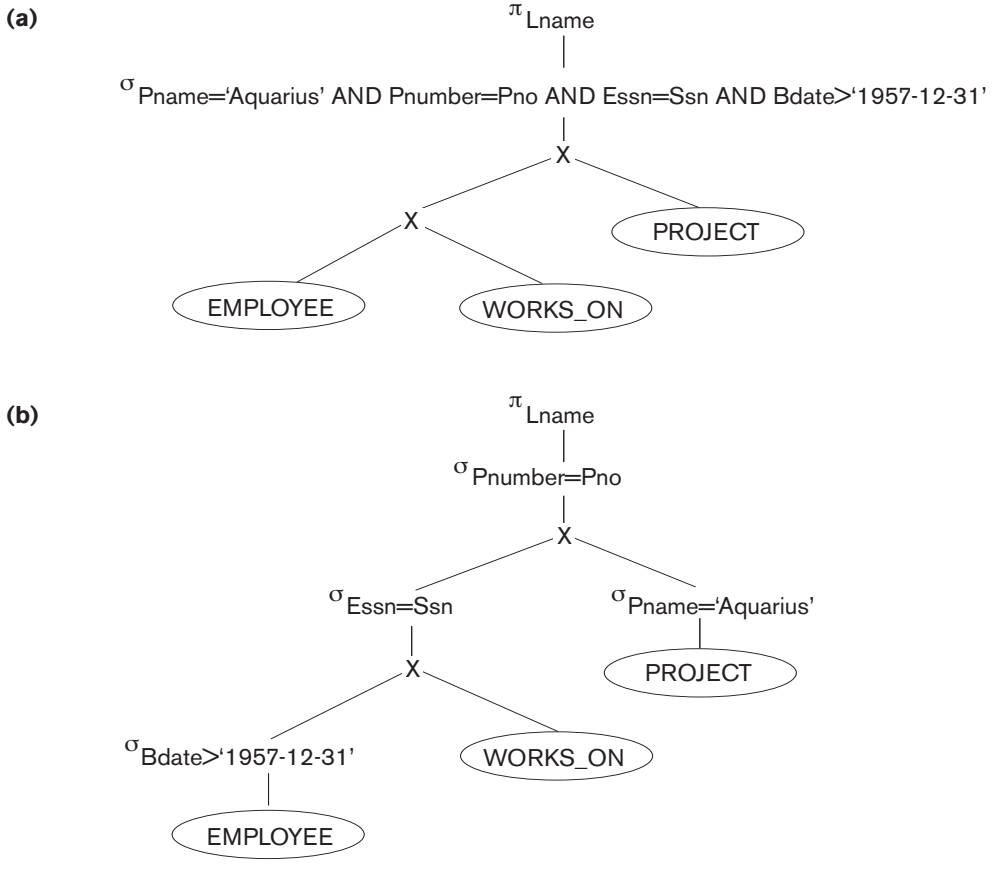
[15]Hence, a query graph corresponds to a *relational calculus* expression as shown in Section 6.6.5.

[16]The same query may also be stated in various ways in a high-level query language such as SQL (see Chapters 4 and 5).

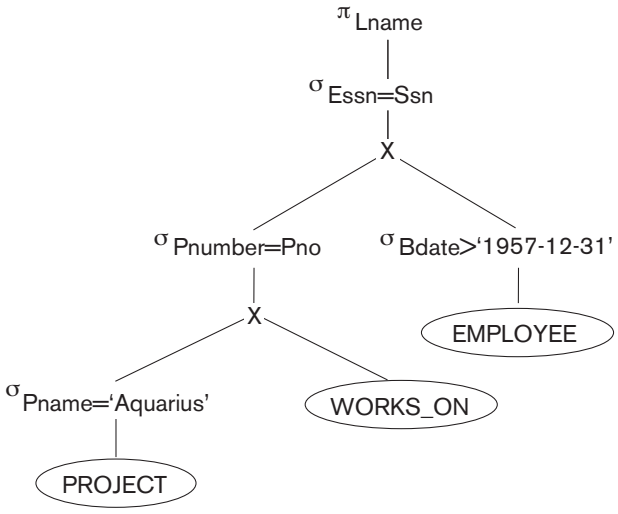
```
Q:  SELECT  Lname
    FROM    EMPLOYEE, WORKS_ON, PROJECT
    WHERE   Pname='Aquarius' AND Pnumber=Pno AND Essn=Ssn
            AND Bdate > '1957-12-31';
```

The initial query tree for Q is shown in Figure 19.5(a). Executing this tree directly first creates a very large file containing the CARTESIAN PRODUCT of the entire EMPLOYEE, WORKS_ON, and PROJECT files. That is why the initial query tree is never executed, but is transformed into another equivalent tree that is efficient to

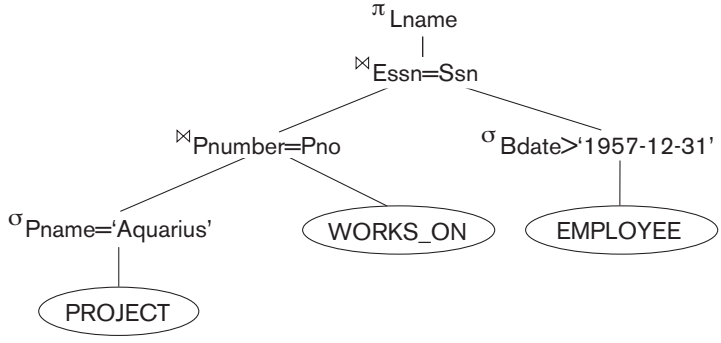
Figure 19.5
Steps in converting a query tree during heuristic optimization.
(a) Initial (canonical) query tree for SQL query Q.
(b) Moving SELECT operations down the query tree.
(c) Applying the more restrictive SELECT operation first.
(d) Replacing CARTESIAN PRODUCT and SELECT with JOIN operations.
(e) Moving PROJECT operations down the query tree.



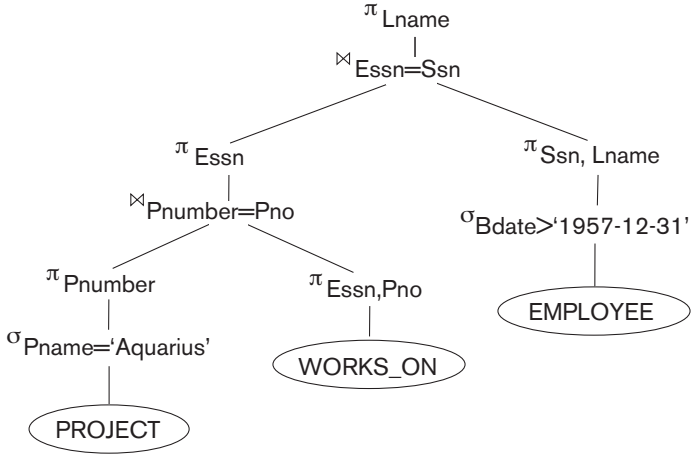
(c)


π Lname
σ Essn=Ssn
X
σ Pnumber=Pno
σ Bdate>'1957-12-31'
X
EMPLOYEE
σ Pname='Aquarius'
WORKS_ON
PROJECT

(d)


π Lname
⋈ Essn=Ssn
⋈ Pnumber=Pno
σ Bdate>'1957-12-31'
σ Pname='Aquarius'
WORKS_ON
EMPLOYEE
PROJECT

(e)


π Lname
⋈ Essn=Ssn
π Essn
π Ssn, Lname
⋈ Pnumber=Pno
σ Bdate>'1957-12-31'
π Pnumber
π Essn,Pno
EMPLOYEE
σ Pname='Aquarius'
WORKS_ON
PROJECT

execute. This particular query needs only one record from the PROJECT relation—for the 'Aquarius' project—and only the EMPLOYEE records for those whose date of birth is after '1957-12-31'. Figure 19.5(b) shows an improved query tree that first applies the SELECT operations to reduce the number of tuples that appear in the CARTESIAN PRODUCT.

A further improvement is achieved by switching the positions of the EMPLOYEE and PROJECT relations in the tree, as shown in Figure 19.5(c). This uses the information that Pnumber is a key attribute of the PROJECT relation, and hence the SELECT operation on the PROJECT relation will retrieve a single record only. We can further improve the query tree by replacing any CARTESIAN PRODUCT operation that is followed by a join condition with a JOIN operation, as shown in Figure 19.5(d). Another improvement is to keep only the attributes needed by subsequent operations in the intermediate relations, by including PROJECT (π) operations as early as possible in the query tree, as shown in Figure 19.5(e). This reduces the attributes (columns) of the intermediate relations, whereas the SELECT operations reduce the number of tuples (records).

As the preceding example demonstrates, a query tree can be transformed step by step into an equivalent query tree that is more efficient to execute. However, we must make sure that the transformation steps always lead to an equivalent query tree. To do this, the query optimizer must know which transformation rules *preserve this equivalence*. We discuss some of these transformation rules next.

General Transformation Rules for Relational Algebra Operations. There are many rules for transforming relational algebra operations into equivalent ones. For query optimization purposes, we are interested in the meaning of the operations and the resulting relations. Hence, if two relations have the same set of attributes in a *different order* but the two relations represent the same information, we consider the relations to be equivalent. In Section 3.1.2 we gave an alternative definition of *relation* that makes the order of attributes unimportant; we will use this definition here. We will state some transformation rules that are useful in query optimization, without proving them:

1. **Cascade of σ** A conjunctive selection condition can be broken up into a cascade (that is, a sequence) of individual σ operations:
   $\sigma_{c_1 \text{ AND } c_2 \text{ AND } \ldots \text{ AND } c_n}(R) \equiv \sigma_{c_1}(\sigma_{c_2}(\ldots(\sigma_{c_n}(R))\ldots))$
2. **Commutativity of σ.** The σ operation is commutative:
   $\sigma_{c_1}(\sigma_{c_2}(R)) \equiv \sigma_{c_2}(\sigma_{c_1}(R))$
3. **Cascade of π.** In a cascade (sequence) of π operations, all but the last one can be ignored:
   $\pi_{\text{List}_1}(\pi_{\text{List}_2}(\ldots(\pi_{\text{List}_n}(R))\ldots)) \equiv \pi_{\text{List}_1}(R)$
4. **Commuting σ with π.** If the selection condition c involves only those attributes $A_1, \ldots, A_n$ in the projection list, the two operations can be commuted:
   $\pi_{A_1, A_2, \ldots, A_n}(\sigma_c(R)) \equiv \sigma_c(\pi_{A_1, A_2, \ldots, A_n}(R))$

5. **Commutativity of $\bowtie$ (and $\times$).** The join operation is commutative, as is the $\times$ operation:

   $R \bowtie_c S \equiv S \bowtie_c R$

   $R \times S \equiv S \times R$

   Notice that although the order of attributes may not be the same in the relations resulting from the two joins (or two Cartesian products), the *meaning* is the same because the order of attributes is not important in the alternative definition of relation.

6. **Commuting σ with $\bowtie$ (or $\times$).** If all the attributes in the selection condition c involve only the attributes of one of the relations being joined—say, R—the two operations can be commuted as follows:

   $\sigma_c (R \bowtie S) \equiv (\sigma_c (R)) \bowtie S$

   Alternatively, if the selection condition c can be written as $(c_1 \text{ AND } c_2)$, where condition c_1 involves only the attributes of R and condition c_2 involves only the attributes of S, the operations commute as follows:

   $\sigma_c (R \bowtie S) \equiv (\sigma_{c_1} (R)) \bowtie (\sigma_{c_2} (S))$

   The same rules apply if the $\bowtie$ is replaced by a $\times$ operation.

7. **Commuting π with $\bowtie$ (or $\times$).** Suppose that the projection list is $L = \{A_1, ..., A_n, B_1, ..., B_m\}$, where $A_1, ..., A_n$ are attributes of R and $B_1, ..., B_m$ are attributes of S. If the join condition c involves only attributes in L, the two operations can be commuted as follows:

   $\pi_L (R \bowtie_c S) \equiv (\pi_{A_1, ..., A_n} (R)) \bowtie_c (\pi_{B_1, ..., B_m} (S))$

   If the join condition c contains additional attributes not in L, these must be added to the projection list, and a final π operation is needed. For example, if attributes $A_{n+1}, ..., A_{n+k}$ of R and $B_{m+1}, ..., B_{m+p}$ of S are involved in the join condition c but are not in the projection list L, the operations commute as follows:

   $\pi_L (R \bowtie_c S) \equiv \pi_L ((\pi_{A_1, ..., A_n, A_{n+1}, ..., A_{n+k}} (R)) \bowtie_c (\pi_{B_1, ..., B_m, B_{m+1}, ..., B_{m+p}} (S)))$

   For $\times$, there is no condition c, so the first transformation rule always applies by replacing $\bowtie_c$ with $\times$.

8. **Commutativity of set operations.** The set operations $\cup$ and $\cap$ are commutative but $-$ is not.

9. **Associativity of $\bowtie$, $\times$, $\cup$, and $\cap$.** These four operations are individually associative; that is, if θ stands for any one of these four operations (throughout the expression), we have:

   $(R \theta S) \theta T \equiv R \theta (S \theta T)$

10. **Commuting σ with set operations.** The σ operation commutes with $\cup$, $\cap$, and $-$. If θ stands for any one of these three operations (throughout the expression), we have:

    $\sigma_c (R \theta S) \equiv (\sigma_c (R)) \theta (\sigma_c (S))$

11. **The π operation commutes with $\cup$.**

    $\pi_L (R \cup S) \equiv (\pi_L (R)) \cup (\pi_L (S))$

12. **Converting a $(\sigma, \times)$ sequence into $\bowtie$.** If the condition c of a σ that follows a $\times$ corresponds to a join condition, convert the $(\sigma, \times)$ sequence into a $\bowtie$ as follows:

    $(\sigma_c (R \times S)) \equiv (R \bowtie_c S)$

There are other possible transformations. For example, a selection or join condition c can be converted into an equivalent condition by using the following standard rules from Boolean algebra (DeMorgan's laws):

NOT $(c_1$ **AND** $c_2) \equiv$ (**NOT** c_1) **OR** (**NOT** c_2)

NOT $(c_1$ **OR** $c_2) \equiv$ (**NOT** c_1) **AND** (**NOT** c_2)

Additional transformations discussed in Chapters 4, 5, and 6 are not repeated here. We discuss next how transformations can be used in heuristic optimization.

Outline of a Heuristic Algebraic Optimization Algorithm. We can now outline the steps of an algorithm that utilizes some of the above rules to transform an initial query tree into a final tree that is more efficient to execute (in most cases). The algorithm will lead to transformations similar to those discussed in our example in Figure 19.5. The steps of the algorithm are as follows:

1. Using Rule 1, break up any SELECT operations with conjunctive conditions into a cascade of SELECT operations. This permits a greater degree of freedom in moving SELECT operations down different branches of the tree.
2. Using Rules 2, 4, 6, and 10 concerning the commutativity of SELECT with other operations, move each SELECT operation as far down the query tree as is permitted by the attributes involved in the select condition. If the condition involves attributes from *only one table*, which means that it represents a *selection condition*, the operation is moved all the way to the leaf node that represents this table. If the condition involves attributes from *two tables*, which means that it represents a *join condition*, the condition is moved to a location down the tree after the two tables are combined.
3. Using Rules 5 and 9 concerning commutativity and associativity of binary operations, rearrange the leaf nodes of the tree using the following criteria. First, position the leaf node relations with the most restrictive SELECT operations so they are executed first in the query tree representation. The definition of *most restrictive* SELECT can mean either the ones that produce a relation with the fewest tuples or with the smallest absolute size.[17] Another possibility is to define the most restrictive SELECT as the one with the smallest selectivity; this is more practical because estimates of selectivities are often available in the DBMS catalog. Second, make sure that the ordering of leaf nodes does not cause CARTESIAN PRODUCT operations; for example, if

[17]Either definition can be used, since these rules are heuristic.

the two relations with the most restrictive SELECT do not have a direct join condition between them, it may be desirable to change the order of leaf nodes to avoid Cartesian products.[18]

4. Using Rule 12, combine a CARTESIAN PRODUCT operation with a subsequent SELECT operation in the tree into a JOIN operation, if the condition represents a join condition.
5. Using Rules 3, 4, 7, and 11 concerning the cascading of PROJECT and the commuting of PROJECT with other operations, break down and move lists of projection attributes down the tree as far as possible by creating new PROJECT operations as needed. Only those attributes needed in the query result and in subsequent operations in the query tree should be kept after each PROJECT operation.
6. Identify subtrees that represent groups of operations that can be executed by a single algorithm.

In our example, Figure 19.5(b) shows the tree in Figure 19.5(a) after applying steps 1 and 2 of the algorithm; Figure 19.5(c) shows the tree after step 3; Figure 19.5(d) after step 4; and Figure 19.5(e) after step 5. In step 6 we may group together the operations in the subtree whose root is the operation π_{Essn} into a single algorithm. We may also group the remaining operations into another subtree, where the tuples resulting from the first algorithm replace the subtree whose root is the operation π_{Essn}, because the first grouping means that this subtree is executed first.

Summary of Heuristics for Algebraic Optimization. The main heuristic is to apply first the operations that reduce the size of intermediate results. This includes performing as early as possible SELECT operations to reduce the number of tuples and PROJECT operations to reduce the number of attributes—by moving SELECT and PROJECT operations as far down the tree as possible. Additionally, the SELECT and JOIN operations that are most restrictive—that is, result in relations with the fewest tuples or with the smallest absolute size—should be executed before other similar operations. The latter rule is accomplished through reordering the leaf nodes of the tree among themselves while avoiding Cartesian products, and adjusting the rest of the tree appropriately.

19.7.3 Converting Query Trees into Query Execution Plans

An execution plan for a relational algebra expression represented as a query tree includes information about the access methods available for each relation as well as the algorithms to be used in computing the relational operators represented in the tree. As a simple example, consider query Q1 from Chapter 4, whose corresponding relational algebra expression is

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

[18]Note that a CARTESIAN PRODUCT is acceptable in some cases—for example, if each relation has only a single tuple because each had a previous select condition on a key field.

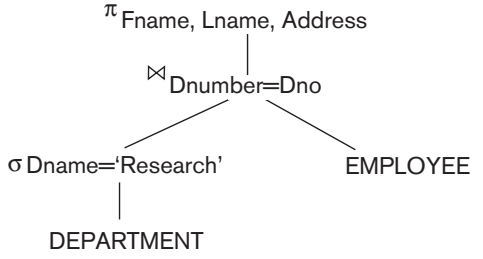


Figure 19.6
A query tree for query Q1.

The query tree is shown in Figure 19.6. To convert this into an execution plan, the optimizer might choose an index search for the SELECT operation on DEPARTMENT (assuming one exists), a single-loop join algorithm that loops over the records in the result of the SELECT operation on DEPARTMENT for the join operation (assuming an index exists on the Dno attribute of EMPLOYEE), and a scan of the JOIN result for input to the PROJECT operator. Additionally, the approach taken for executing the query may specify a materialized or a pipelined evaluation, although in general a pipelined evaluation is preferred whenever feasible.

With **materialized evaluation**, the result of an operation is stored as a temporary relation (that is, the result is *physically materialized*). For instance, the JOIN operation can be computed and the entire result stored as a temporary relation, which is then read as input by the algorithm that computes the PROJECT operation, which would produce the query result table. On the other hand, with **pipelined evaluation**, as the resulting tuples of an operation are produced, they are forwarded directly to the next operation in the query sequence. For example, as the selected tuples from DEPARTMENT are produced by the SELECT operation, they are placed in a buffer; the JOIN operation algorithm would then consume the tuples from the buffer, and those tuples that result from the JOIN operation are pipelined to the projection operation algorithm. The advantage of pipelining is the cost savings in not having to write the intermediate results to disk and not having to read them back for the next operation.

19.8 Using Selectivity and Cost Estimates in Query Optimization

A query optimizer does not depend solely on heuristic rules; it also estimates and compares the costs of executing a query using different execution strategies and algorithms, and it then chooses the strategy with the *lowest cost estimate*. For this approach to work, accurate *cost estimates* are required so that different strategies can be compared fairly and realistically. In addition, the optimizer must limit the number of execution strategies to be considered; otherwise, too much time will be spent making cost estimates for the many possible execution strategies. Hence, this approach is more suitable for **compiled queries** where the optimization is done at compile time and the resulting execution strategy code is stored and executed directly at runtime. For **interpreted queries**, where the entire process shown in

Figure 19.1 occurs at runtime, a full-scale optimization may slow down the response time. A more elaborate optimization is indicated for compiled queries, whereas a partial, less time-consuming optimization works best for interpreted queries.

This approach is generally referred to as **cost-based query optimization**.[19] It uses traditional optimization techniques that search the *solution space* to a problem for a solution that minimizes an objective (cost) function. The cost functions used in query optimization are estimates and not exact cost functions, so the optimization may select a query execution strategy that is not the optimal (absolute best) one. In Section 19.8.1 we discuss the components of query execution cost. In Section 19.8.2 we discuss the type of information needed in cost functions. This information is kept in the DBMS catalog. In Section 19.8.3 we give examples of cost functions for the SELECT operation, and in Section 19.8.4 we discuss cost functions for two-way JOIN operations. Section 19.8.5 discusses multiway joins, and Section 19.8.6 gives an example.

19.8.1 Cost Components for Query Execution

The cost of executing a query includes the following components:

1. **Access cost to secondary storage.** This is the cost of transferring (reading and writing) data blocks between secondary disk storage and main memory buffers. This is also known as *disk I/O (input/output) cost*. The cost of searching for records in a disk file depends on the type of access structures on that file, such as ordering, hashing, and primary or secondary indexes. In addition, factors such as whether the file blocks are allocated contiguously on the same disk cylinder or scattered on the disk affect the access cost.
2. **Disk storage cost.** This is the cost of storing on disk any intermediate files that are generated by an execution strategy for the query.
3. **Computation cost.** This is the cost of performing in-memory operations on the records within the data buffers during query execution. Such operations include searching for and sorting records, merging records for a join or a sort operation, and performing computations on field values. This is also known as *CPU (central processing unit) cost*.
4. **Memory usage cost.** This is the cost pertaining to the number of main memory buffers needed during query execution.
5. **Communication cost.** This is the cost of shipping the query and its results from the database site to the site or terminal where the query originated. In distributed databases (see Chapter 25), it would also include the cost of transferring tables and results among various computers during query evaluation.

For large databases, the main emphasis is often on minimizing the access cost to secondary storage. Simple cost functions ignore other factors and compare different query execution strategies in terms of the number of block transfers between disk

[19]This approach was first used in the optimizer for the SYSTEM R in an experimental DBMS developed at IBM (Selinger et al. 1979).

and main memory buffers. For smaller databases, where most of the data in the files involved in the query can be completely stored in memory, the emphasis is on minimizing computation cost. In distributed databases, where many sites are involved (see Chapter 25), communication cost must be minimized also. It is difficult to include all the cost components in a (weighted) cost function because of the difficulty of assigning suitable weights to the cost components. That is why some cost functions consider a single factor only—disk access. In the next section we discuss some of the information that is needed for formulating cost functions.

19.8.2 Catalog Information Used in Cost Functions

To estimate the costs of various execution strategies, we must keep track of any information that is needed for the cost functions. This information may be stored in the DBMS catalog, where it is accessed by the query optimizer. First, we must know the size of each file. For a file whose records are all of the same type, the **number of records (tuples)** (***r***), the (average) **record size** (***R***), and the **number of file blocks** (***b***) (or close estimates of them) are needed. The **blocking factor** (***bfr***) for the file may also be needed. We must also keep track of the *primary file organization* for each file. The primary file organization records may be *unordered*, *ordered* by an attribute with or without a primary or clustering index, or *hashed* (static hashing or one of the dynamic hashing methods) on a key attribute. Information is also kept on all primary, secondary, or clustering indexes and their indexing attributes. The **number of levels** (***x***) of each multilevel index (primary, secondary, or clustering) is needed for cost functions that estimate the number of block accesses that occur during query execution. In some cost functions the **number of first-level index blocks** (b_{I1}) is needed.

Another important parameter is the **number of distinct values** (***d***) of an attribute and the attribute **selectivity** (***sl***), which is the fraction of records satisfying an equality condition on the attribute. This allows estimation of the **selection cardinality** (***s*** = ***sl*** * ***r***) of an attribute, which is the *average* number of records that will satisfy an equality selection condition on that attribute. For a *key attribute,* $d = r$, $sl = 1/r$ and $s = 1$. For a *nonkey attribute,* by making an assumption that the d distinct values are uniformly distributed among the records, we estimate $sl = (1/d)$ and so $s = (r/d)$.[20]

Information such as the number of index levels is easy to maintain because it does not change very often. However, other information may change frequently; for example, the number of records r in a file changes every time a record is inserted or deleted. The query optimizer will need reasonably close but not necessarily completely up-to-the-minute values of these parameters for use in estimating the cost of various execution strategies.

For a nonkey attribute with d distinct values, it is often the case that the records are not uniformly distributed among these values. For example, suppose that a company has 5 departments numbered 1 through 5, and 200 employees who are distrib-

[20]More accurate optimizers store *histograms* of the distribution of records over the data values for an attribute.

uted among the departments as follows: (1, 5), (2, 25), (3, 70), (4, 40), (5, 60). In such cases, the optimizer can store a **histogram** that reflects the distribution of employee records over different departments in a table with the two attributes (Dno, Selectivity), which would contain the following values for our example: (1, 0.025), (2, 0.125), (3, 0.35), (4, 0.2), (5, 0.3). The selectivity values stored in the histogram can also be estimates if the employee table changes frequently.

In the next two sections we examine how some of these parameters are used in cost functions for a cost-based query optimizer.

19.8.3 Examples of Cost Functions for SELECT

We now give cost functions for the selection algorithms S1 to S8 discussed in Section 19.3.1 in terms of *number of block transfers* between memory and disk. Algorithm S9 involves an intersection of record pointers after they have been retrieved by some other means, such as algorithm S6, and so the cost function will be based on the cost for S6. These cost functions are estimates that ignore computation time, storage cost, and other factors. The cost for method S*i* is referred to as C_{Si} block accesses.

- **S1—Linear search (brute force) approach.** We search all the file blocks to retrieve all records satisfying the selection condition; hence, $C_{S1a} = b$. For an *equality condition on a key attribute*, only half the file blocks are searched *on the average* before finding the record, so a rough estimate for $C_{S1b} = (b/2)$ if the record is found; if no record is found that satisfies the condition, $C_{S1b} = b$.
- **S2—Binary search.** This search accesses approximately $C_{S2} = \log_2 b + \lceil (s/bfr) \rceil - 1$ file blocks. This reduces to $\log_2 b$ if the equality condition is on a unique (key) attribute, because $s = 1$ in this case.
- **S3a—Using a primary index to retrieve a single record.** For a primary index, retrieve one disk block at each index level, plus one disk block from the data file. Hence, the cost is one more disk block than the number of index levels: $C_{S3a} = x + 1$.
- **S3b—Using a hash key to retrieve a single record.** For hashing, only one disk block needs to be accessed in most cases. The cost function is approximately $C_{S3b} = 1$ for static hashing or linear hashing, and it is 2 disk block accesses for extendible hashing (see Section 17.8).
- **S4—Using an ordering index to retrieve multiple records.** If the comparison condition is >, >=, <, or <= on a key field with an ordering index, roughly half the file records will satisfy the condition. This gives a cost function of $C_{S4} = x + (b/2)$. This is a very rough estimate, and although it may be correct on the average, it may be quite inaccurate in individual cases. A more accurate estimate is possible if the distribution of records is stored in a histogram.
- **S5—Using a clustering index to retrieve multiple records.** One disk block is accessed at each index level, which gives the address of the first file disk block in the cluster. Given an equality condition on the indexing attribute, *s*

records will satisfy the condition, where s is the selection cardinality of the indexing attribute. This means that $\lceil (s/bfr) \rceil$ file blocks will be in the cluster of file blocks that hold all the selected records, giving $C_{S5} = x + \lceil (s/bfr) \rceil$.

- **S6—Using a secondary (B^+-tree) index.** For a secondary index on a key (unique) attribute, the cost is $x + 1$ disk block accesses. For a secondary index on a nonkey (nonunique) attribute, s records will satisfy an equality condition, where s is the selection cardinality of the indexing attribute. However, because the index is nonclustering, each of the records may reside on a different disk block, so the (worst case) cost estimate is $C_{S6a} = x + 1 + s$. The additional *1* is to account for the disk block that contains the record pointers after the index is searched (see Figure 18.5). If the comparison condition is >, >=, <, or <= and half the file records are assumed to satisfy the condition, then (very roughly) half the first-level index blocks are accessed, plus half the file records via the index. The cost estimate for this case, approximately, is $C_{S6b} = x + (b_{I1}/2) + (r/2)$. The $r/2$ factor can be refined if better selectivity estimates are available through a histogram. The latter method C_{S6b} can be very costly.
- **S7—Conjunctive selection.** We can use either S1 or one of the methods S2 to S6 discussed above. In the latter case, we use one condition to retrieve the records and then check in the main memory buffers whether each retrieved record satisfies the remaining conditions in the conjunction. If multiple indexes exist, the search of each index can produce a set of record pointers (record ids) in the main memory buffers. The intersection of the sets of record pointers (referred to in S9) can be computed in main memory, and then the resulting records are retrieved based on their record ids.
- **S8—Conjunctive selection using a composite index.** Same as S3*a*, S5, or S6*a*, depending on the type of index.

Example of Using the Cost Functions. In a query optimizer, it is common to enumerate the various possible strategies for executing a query and to estimate the costs for different strategies. An optimization technique, such as dynamic programming, may be used to find the optimal (least) cost estimate efficiently, without having to consider all possible execution strategies. We do not discuss optimization algorithms here; rather, we use a simple example to illustrate how cost estimates may be used. Suppose that the EMPLOYEE file in Figure 3.5 has $r_E = 10{,}000$ records stored in $b_E = 2000$ disk blocks with blocking factor $bfr_E = 5$ records/block and the following access paths:

1. A clustering index on Salary, with levels $x_{\text{Salary}} = 3$ and average selection cardinality $s_{\text{Salary}} = 20$. (This corresponds to a selectivity of $sl_{\text{Salary}} = 0.002$).
2. A secondary index on the key attribute Ssn, with $x_{\text{Ssn}} = 4$ ($s_{\text{Ssn}} = 1$, $sl_{\text{Ssn}} = 0.0001$).
3. A secondary index on the nonkey attribute Dno, with $x_{\text{Dno}} = 2$ and first-level index blocks $b_{I1\text{Dno}} = 4$. There are $d_{\text{Dno}} = 125$ distinct values for Dno, so the selectivity of Dno is $sl_{\text{Dno}} = (1/d_{\text{Dno}}) = 0.008$, and the selection cardinality is $s_{\text{Dno}} = (r_E * sl_{\text{Dno}}) = (r_E/d_{\text{Dno}}) = 80$.

4. A secondary index on Sex, with $x_{\text{Sex}} = 1$. There are $d_{\text{Sex}} = 2$ values for the Sex attribute, so the average selection cardinality is $s_{\text{Sex}} = (r_E/d_{\text{Sex}}) = 5000$. (Note that in this case, a histogram giving the percentage of male and female employees may be useful, unless they are approximately equal.)

We illustrate the use of cost functions with the following examples:

OP1: $\sigma_{\text{Ssn='123456789'}}(\text{EMPLOYEE})$

OP2: $\sigma_{\text{Dno}>5}(\text{EMPLOYEE})$

OP3: $\sigma_{\text{Dno}=5}(\text{EMPLOYEE})$

OP4: $\sigma_{\text{Dno}=5 \text{ AND SALARY}>30000 \text{ AND Sex='F'}}(\text{EMPLOYEE})$

The cost of the brute force (linear search or file scan) option S1 will be estimated as $C_{S1a} = b_E = 2000$ (for a selection on a nonkey attribute) or $C_{S1b} = (b_E/2) = 1000$ (average cost for a selection on a key attribute). For OP1 we can use either method *S*1 or method *S6a*; the cost estimate for *S6a* is $C_{S6a} = x_{\text{Ssn}} + 1 = 4 + 1 = 5$, and it is chosen over method *S*1, whose average cost is $C_{S1b} = 1000$. For OP2 we can use either method *S*1 (with estimated cost $C_{S1a} = 2000$) or method S6*b* (with estimated cost $C_{S6b} = x_{\text{Dno}} + (b_{I1\text{Dno}}/2) + (r_E/2) = 2 + (4/2) + (10{,}000/2) = 5004$), so we choose the linear search approach for OP2. For OP3 we can use either method *S*1 (with estimated cost $C_{S1a} = 2000$) or method *S6a* (with estimated cost $C_{S6a} = x_{\text{Dno}} + s_{\text{Dno}} = 2 + 80 = 82$), so we choose method *S6a*.

Finally, consider OP4, which has a conjunctive selection condition. We need to estimate the cost of using any one of the three components of the selection condition to retrieve the records, plus the linear search approach. The latter gives cost estimate $C_{S1a} = 2000$. Using the condition (Dno = 5) first gives the cost estimate $C_{S6a} = 82$. Using the condition (Salary > 30,000) first gives a cost estimate $C_{S4} = x_{\text{Salary}} + (b_E/2) = 3 + (2000/2) = 1003$. Using the condition (Sex = 'F') first gives a cost estimate $C_{S6a} = x_{\text{Sex}} + s_{\text{Sex}} = 1 + 5000 = 5001$. The optimizer would then choose method *S6a* on the secondary index on Dno because it has the lowest cost estimate. The condition (Dno = 5) is used to retrieve the records, and the remaining part of the conjunctive condition (Salary > 30,000 AND Sex = 'F') is checked for each selected record after it is retrieved into memory. Only the records that satisfy these additional conditions are included in the result of the operation.

19.8.4 Examples of Cost Functions for JOIN

To develop reasonably accurate cost functions for JOIN operations, we need to have an estimate for the size (number of tuples) of the file that results *after* the JOIN operation. This is usually kept as a ratio of the size (number of tuples) of the resulting join file to the size of the CARTESIAN PRODUCT file, if both are applied to the same input files, and it is called the **join selectivity** (***js***). If we denote the number of tuples of a relation *R* by |*R*|, we have:

$$js = |(R \bowtie_c S)| / |(R \times S)| = |(R \bowtie_c S)| / (|R| * |S|)$$

If there is no join condition *c,* then *js* = 1 and the join is the same as the CARTESIAN PRODUCT. If no tuples from the relations satisfy the join condition, then *js* = 0. In

general, $0 \le js \le 1$. For a join where the condition c is an equality comparison $R.A = S.B$, we get the following two special cases:

1. If A is a key of R, then $|(R \bowtie_c S)| \le |S|$, so $js \le (1/|R|)$. This is because each record in file S will be joined with at most one record in file R, since A is a key of R. A special case of this condition is when attribute B is a *foreign key* of S that references the *primary key* A of R. In addition, if the foreign key B has the NOT NULL constraint, then $js = (1/|R|)$, and the result file of the join will contain $|S|$ records.
2. If B is a key of S, then $|(R \bowtie_c S)| \le |R|$, so $js \le (1/|S|)$.

Having an estimate of the join selectivity for commonly occurring join conditions enables the query optimizer to estimate the size of the resulting file after the join operation, given the sizes of the two input files, by using the formula $|(R \bowtie_c S)| = js * |R| * |S|$. We can now give some sample *approximate* cost functions for estimating the cost of some of the join algorithms given in Section 19.3.2. The join operations are of the form:

$$R \bowtie_{A=B} S$$

where A and B are domain-compatible attributes of R and S, respectively. Assume that R has b_R blocks and that S has b_S blocks:

- **J1—Nested-loop join.** Suppose that we use R for the outer loop; then we get the following cost function to estimate the number of block accesses for this method, assuming *three memory buffers.* We assume that the blocking factor for the resulting file is bfr_{RS} and that the join selectivity is known:

  $C_{J1} = b_R + (b_R * b_S) + ((js * |R| * |S|)/bfr_{RS})$

  The last part of the formula is the cost of writing the resulting file to disk. This cost formula can be modified to take into account different numbers of memory buffers, as presented in Section 19.3.2. If n_B main memory buffers are available to perform the join, the cost formula becomes:

  $C_{J1} = b_R + (\lceil b_R/(n_B - 2) \rceil * b_S) + ((js * |R| * |S|)/bfr_{RS})$
- **J2—Single-loop join (using an access structure to retrieve the matching record(s)).** If an index exists for the join attribute B of S with index levels x_B, we can retrieve each record s in R and then use the index to retrieve all the matching records t from S that satisfy $t[B] = s[A]$. The cost depends on the type of index. For a secondary index where s_B is the selection cardinality for the join attribute B of S,[21] we get:

  $C_{J2a} = b_R + (|R| * (x_B + 1 + s_B)) + ((js * |R| * |S|)/bfr_{RS})$

  For a clustering index where s_B is the selection cardinality of B, we get

  $C_{J2b} = b_R + (|R| * (x_B + (s_B/bfr_B))) + ((js * |R| * |S|)/bfr_{RS})$

  For a primary index, we get

[21] *Selection cardinality* was defined as the average number of records that satisfy an equality condition on an attribute, which is the average number of records that have the same value for the attribute and hence will be joined to a single record in the other file.

$C_{J2c} = b_R + (|R| * (x_B + 1)) + ((js * |R| * |S|)/bfr_{RS})$

If a hash key exists for one of the two join attributes—say, B of S—we get

$C_{J2d} = b_R + (|R| * h) + ((js * |R| * |S|)/bfr_{RS})$

where $h \geq 1$ is the average number of block accesses to retrieve a record, given its hash key value. Usually, h is estimated to be *1* for static and linear hashing and *2* for extendible hashing.

- **J3—Sort-merge join.** If the files are already sorted on the join attributes, the cost function for this method is

  $C_{J3a} = b_R + b_S + ((js * |R| * |S|)/bfr_{RS})$

  If we must sort the files, the cost of sorting must be added. We can use the formulas from Section 19.2 to estimate the sorting cost.

Example of Using the Cost Functions. Suppose that we have the EMPLOYEE file described in the example in the previous section, and assume that the DEPARTMENT file in Figure 3.5 consists of $r_D = 125$ records stored in $b_D = 13$ disk blocks. Consider the following two join operations:

OP6: EMPLOYEE $\bowtie_{\text{Dno=Dnumber}}$ DEPARTMENT
OP7: DEPARTMENT $\bowtie_{\text{Mgr\_ssn=Ssn}}$ EMPLOYEE

Suppose that we have a primary index on Dnumber of DEPARTMENT with $x_{\text{Dnumber}} = 1$ level and a secondary index on Mgr_ssn of DEPARTMENT with selection cardinality $s_{\text{Mgr\_ssn}} = 1$ and levels $x_{\text{Mgr\_ssn}} = 2$. Assume that the join selectivity for OP6 is $js_{\text{OP6}} = (1/|\text{DEPARTMENT}|) = 1/125$ because Dnumber is a key of DEPARTMENT. Also assume that the blocking factor for the resulting join file is $bfr_{\text{ED}} = 4$ records per block. We can estimate the worst-case costs for the JOIN operation OP6 using the applicable methods J1 and J2 as follows:

1. Using method J1 with EMPLOYEE as outer loop:

   $C_{J1} = b_E + (b_E * b_D) + ((js_{\text{OP6}} * r_E * r_D)/bfr_{ED})$
   $= 2000 + (2000 * 13) + (((1/125) * 10{,}000 * 125)/4) = 30{,}500$

2. Using method J1 with DEPARTMENT as outer loop:

   $C_{J1} = b_D + (b_E * b_D) + ((js_{\text{OP6}} * r_E * r_D)/bfr_{ED})$
   $= 13 + (13 * 2000) + (((1/125) * 10{,}000 * 125/4) = 28{,}513$

3. Using method J2 with EMPLOYEE as outer loop:

   $C_{J2c} = b_E + (r_E * (x_{\text{Dnumber}} + 1)) + ((js_{\text{OP6}} * r_E * r_D)/bfr_{ED}$
   $= 2000 + (10{,}000 * 2) + (((1/125) * 10{,}000 * 125/4) = 24{,}500$

4. Using method J2 with DEPARTMENT as outer loop:

   $C_{J2a} = b_D + (r_D * (x_{\text{Dno}} + s_{\text{Dno}})) + ((js_{\text{OP6}} * r_E * r_D)/bfr_{ED})$
   $= 13 + (125 * (2 + 80)) + (((1/125) * 10{,}000 * 125/4) = 12{,}763$

Case 4 has the lowest cost estimate and will be chosen. Notice that in case 2 above, if 15 memory buffers (or more) were available for executing the join instead of just 3, 13 of them could be used to hold the entire DEPARTMENT relation (outer loop

relation) in memory, one could be used as buffer for the result, and one would be used to hold one block at a time of the EMPLOYEE file (inner loop file), and the cost for case 2 could be drastically reduced to just $b_E + b_D + ((\ js_{\mathrm{OP6}} * r_{\mathrm{E}} * r_{\mathrm{D}})/bfr_{\mathrm{ED}})$ or 4,513, as discussed in Section 19.3.2. If some other number of main memory buffers was available, say $n_{\mathrm{B}} = 10$, then the cost for case 2 would be calculated as follows, which would also give better performance than case 4:

$$\begin{aligned} C_{\mathrm{J1}} &= b_D + (\lceil b_D/(n_B - 2)\rceil * b_E) + ((js * |R| * |S|)/bfr_{RS}) \\ &= 13 + (\lceil 13/8\rceil * 2000) + (((1/125) * 10{,}000 * 125/4) = 28{,}513 \\ &= 13 + (2 * 2000) + 2500 = 6{,}513 \end{aligned}$$

As an exercise, the reader should perform a similar analysis for OP7.

19.8.5 Multiple Relation Queries and JOIN Ordering

The algebraic transformation rules in Section 19.7.2 include a commutative rule and an associative rule for the join operation. With these rules, many equivalent join expressions can be produced. As a result, the number of alternative query trees grows very rapidly as the number of joins in a query increases. A query that joins n relations will often have $n - 1$ join operations, and hence can have a large number of different join orders. Estimating the cost of every possible join tree for a query with a large number of joins will require a substantial amount of time by the query optimizer. Hence, some pruning of the possible query trees is needed. Query optimizers typically limit the structure of a (join) query tree to that of left-deep (or right-deep) trees. A **left-deep tree** is a binary tree in which the right child of each nonleaf node is always a base relation. The optimizer would choose the particular left-deep tree with the lowest estimated cost. Two examples of left-deep trees are shown in Figure 19.7. (Note that the trees in Figure 19.5 are also left-deep trees.)

With left-deep trees, the right child is considered to be the inner relation when executing a nested-loop join, or the probing relation when executing a single-loop join. One advantage of left-deep (or right-deep) trees is that they are amenable to pipelining, as discussed in Section 19.6. For instance, consider the first left-deep tree in Figure 19.7 and assume that the join algorithm is the single-loop method; in this case, a disk page of tuples of the outer relation is used to probe the inner relation for

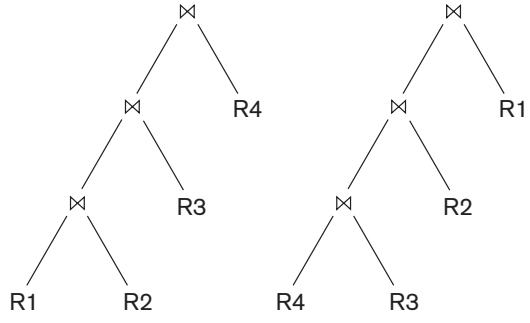


Figure 19.7
Two left-deep (JOIN) query trees.

matching tuples. As resulting tuples (records) are produced from the join of R1 and R2, they can be used to probe R3 to locate their matching records for joining. Likewise, as resulting tuples are produced from this join, they could be used to probe R4. Another advantage of left-deep (or right-deep) trees is that having a base relation as one of the inputs of each join allows the optimizer to utilize any access paths on that relation that may be useful in executing the join.

If materialization is used instead of pipelining (see Sections 19.6 and 19.7.3), the join results could be materialized and stored as temporary relations. The key idea from the optimizer's standpoint with respect to join ordering is to find an ordering that will reduce the size of the temporary results, since the temporary results (pipelined or materialized) are used by subsequent operators and hence affect the execution cost of those operators.

19.8.6 Example to Illustrate Cost-Based Query Optimization

We will consider query Q2 and its query tree shown in Figure 19.4(a) to illustrate cost-based query optimization:

```
Q2: SELECT   Pnumber, Dnum, Lname, Address, Bdate
    FROM     PROJECT, DEPARTMENT, EMPLOYEE
    WHERE    Dnum=Dnumber AND Mgr_ssn=Ssn AND
             Plocation='Stafford';
```

Suppose we have the information about the relations shown in Figure 19.8. The LOW_VALUE and HIGH_VALUE statistics have been normalized for clarity. The tree in Figure 19.4(a) is assumed to represent the result of the algebraic heuristic optimization process and the start of cost-based optimization (in this example, we assume that the heuristic optimizer does not push the projection operations down the tree).

The first cost-based optimization to consider is join ordering. As previously mentioned, we assume the optimizer considers only left-deep trees, so the potential join orders—without CARTESIAN PRODUCT—are:

1. PROJECT ⋈ DEPARTMENT ⋈ EMPLOYEE
2. DEPARTMENT ⋈ PROJECT ⋈ EMPLOYEE
3. DEPARTMENT ⋈ EMPLOYEE ⋈ PROJECT
4. EMPLOYEE ⋈ DEPARTMENT ⋈ PROJECT

Assume that the selection operation has already been applied to the PROJECT relation. If we assume a materialized approach, then a new temporary relation is created after each join operation. To examine the cost of join order (1), the first join is between PROJECT and DEPARTMENT. Both the join method and the access methods for the input relations must be determined. Since DEPARTMENT has no index according to Figure 19.8, the only available access method is a table scan (that is, a linear search). The PROJECT relation will have the selection operation performed before the join, so two options exist: table scan (linear search) or utilizing its PROJ_PLOC index, so the optimizer must compare their estimated costs.

Figure 19.8
Sample statistical information for relations in Q2. (a) Column information. (b) Table information. (c) Index information.

(a)

Table_name	Column_name	Num_distinct	Low_value	High_value
PROJECT	Plocation	200	1	200
PROJECT	Pnumber	2000	1	2000
PROJECT	Dnum	50	1	50
DEPARTMENT	Dnumber	50	1	50
DEPARTMENT	Mgr_ssn	50	1	50
EMPLOYEE	Ssn	10000	1	10000
EMPLOYEE	Dno	50	1	50
EMPLOYEE	Salary	500	1	500

(b)

Table_name	Num_rows	Blocks
PROJECT	2000	100
DEPARTMENT	50	5
EMPLOYEE	10000	2000

(c)

Index_name	Uniqueness	Blevel*	Leaf_blocks	Distinct_keys
PROJ_PLOC	NONUNIQUE	1	4	200
EMP_SSN	UNIQUE	1	50	10000
EMP_SAL	NONUNIQUE	1	50	500

*Blevel is the number of levels without the leaf level.

The statistical information on the PROJ_PLOC index (see Figure 19.8) shows the number of index levels $x = 2$ (root plus leaf levels). The index is nonunique (because Plocation is not a key of PROJECT), so the optimizer assumes a uniform data distribution and estimates the number of record pointers for each Plocation value to be 10. This is computed from the tables in Figure 19.8 by multiplying Selectivity * Num_rows, where Selectivity is estimated by 1/Num_distinct. So the cost of using the index and accessing the records is estimated to be 12 block accesses (2 for the index and 10 for the data blocks). The cost of a table scan is estimated to be 100 block accesses, so the index access is more efficient as expected.

In the materialized approach, a temporary file TEMP1 of size 1 block is created to hold the result of the selection operation. The file size is calculated by determining the blocking factor using the formula Num_rows/Blocks, which gives 2000/100 or 20 rows per block. Hence, the 10 records selected from the PROJECT relation will fit

into a single block. Now we can compute the estimated cost of the first join. We will consider only the nested-loop join method, where the outer relation is the temporary file, TEMP1, and the inner relation is DEPARTMENT. Since the entire TEMP1 file fits in the available buffer space, we need to read each of the DEPARTMENT table's five blocks only once, so the join cost is six block accesses plus the cost of writing the temporary result file, TEMP2. The optimizer would have to determine the size of TEMP2. Since the join attribute Dnumber is the key for DEPARTMENT, any Dnum value from TEMP1 will join with at most one record from DEPARTMENT, so the number of rows in TEMP2 will be equal to the number of rows in TEMP1, which is 10. The optimizer would determine the record size for TEMP2 and the number of blocks needed to store these 10 rows. For brevity, assume that the blocking factor for TEMP2 is five rows per block, so a total of two blocks are needed to store TEMP2.

Finally, the cost of the last join needs to be estimated. We can use a single-loop join on TEMP2 since in this case the index EMP_SSN (see Figure 19.8) can be used to probe and locate matching records from EMPLOYEE. Hence, the join method would involve reading in each block of TEMP2 and looking up each of the five Mgr_ssn values using the EMP_SSN index. Each index lookup would require a root access, a leaf access, and a data block access (x+1, where the number of levels x is 2). So, 10 lookups require 30 block accesses. Adding the two block accesses for TEMP2 gives a total of 32 block accesses for this join.

For the final projection, assume pipelining is used to produce the final result, which does not require additional block accesses, so the total cost for join order (1) is estimated as the sum of the previous costs. The optimizer would then estimate costs in a similar manner for the other three join orders and choose the one with the lowest estimate. We leave this as an exercise for the reader.

19.9 Overview of Query Optimization in Oracle

The Oracle DBMS[22] provides two different approaches to query optimization: rule-based and cost-based. With the rule-based approach, the optimizer chooses execution plans based on heuristically ranked operations. Oracle maintains a table of 15 ranked access paths, where a lower ranking implies a more efficient approach. The access paths range from table access by ROWID (the most efficient)—where ROWID specifies the record's physical address that includes the data file, data block, and row offset within the block—to a full table scan (the least efficient)—where all rows in the table are searched by doing multiblock reads. However, the rule-based approach is being phased out in favor of the cost-based approach, where the optimizer examines alternative access paths and operator algorithms and chooses the execution plan with the lowest estimated cost. The estimated query cost is proportional to the expected elapsed time needed to execute the query with the given execution plan.

[22]The discussion in this section is primarily based on version 7 of Oracle. More optimization techniques have been added to subsequent versions.

The Oracle optimizer calculates this cost based on the estimated usage of resources, such as I/O, CPU time, and memory needed. The goal of cost-based optimization in Oracle is to minimize the elapsed time to process the entire query.

An interesting addition to the Oracle query optimizer is the capability for an application developer to specify **hints** to the optimizer.[23] The idea is that an application developer might know more information about the data than the optimizer. For example, consider the EMPLOYEE table shown in Figure 3.6. The Sex column of that table has only two distinct values. If there are 10,000 employees, then the optimizer would estimate that half are male and half are female, assuming a uniform data distribution. If a secondary index exists, it would more than likely not be used. However, if the application developer knows that there are only 100 male employees, a hint could be specified in an SQL query whose WHERE-clause condition is Sex = 'M' so that the associated index would be used in processing the query. Various hints can be specified, such as:

- The optimization approach for an SQL statement
- The access path for a table accessed by the statement
- The join order for a join statement
- A particular join operation in a join statement

The cost-based optimization of Oracle 8 and later versions is a good example of the sophisticated approach taken to optimize SQL queries in commercial RDBMSs.

19.10 Semantic Query Optimization

A different approach to query optimization, called **semantic query optimization**, has been suggested. This technique, which may be used in combination with the techniques discussed previously, uses constraints specified on the database schema—such as unique attributes and other more complex constraints—in order to modify one query into another query that is more efficient to execute. We will not discuss this approach in detail but we will illustrate it with a simple example. Consider the SQL query:

```
SELECT  E.Lname, M.Lname
FROM    EMPLOYEE AS E, EMPLOYEE AS M
WHERE   E.Super_ssn=M.Ssn AND E.Salary > M.Salary
```

This query retrieves the names of employees who earn more than their supervisors. Suppose that we had a constraint on the database schema that stated that no employee can earn more than his or her direct supervisor. If the semantic query optimizer checks for the existence of this constraint, it does not need to execute the query at all because it knows that the result of the query will be empty. This may save considerable time if the constraint checking can be done efficiently. However, searching through many constraints to find those that are applicable to a given

[23]Such hints have also been called query *annotations*.

query and that may semantically optimize it can also be quite time-consuming. With the inclusion of active rules and additional metadata in database systems (see Chapter 26), semantic query optimization techniques are being gradually incorporated into the DBMSs.

19.11 Summary

In this chapter we gave an overview of the techniques used by DBMSs in processing and optimizing high-level queries. We first discussed how SQL queries are translated into relational algebra and then how various relational algebra operations may be executed by a DBMS. We saw that some operations, particularly SELECT and JOIN, may have many execution options. We also discussed how operations can be combined during query processing to create pipelined or stream-based execution instead of materialized execution.

Following that, we described heuristic approaches to query optimization, which use heuristic rules and algebraic techniques to improve the efficiency of query execution. We showed how a query tree that represents a relational algebra expression can be heuristically optimized by reorganizing the tree nodes and transforming it into another equivalent query tree that is more efficient to execute. We also gave equivalence-preserving transformation rules that may be applied to a query tree. Then we introduced query execution plans for SQL queries, which add method execution plans to the query tree operations.

We discussed the cost-based approach to query optimization. We showed how cost functions are developed for some database access algorithms and how these cost functions are used to estimate the costs of different execution strategies. We presented an overview of the Oracle query optimizer, and we mentioned the technique of semantic query optimization.

Review Questions

19.1. Discuss the reasons for converting SQL queries into relational algebra queries before optimization is done.

19.2. Discuss the different algorithms for implementing each of the following relational operators and the circumstances under which each algorithm can be used: SELECT, JOIN, PROJECT, UNION, INTERSECT, SET DIFFERENCE, CARTESIAN PRODUCT.

19.3. What is a query execution plan?

19.4. What is meant by the term *heuristic optimization*? Discuss the main heuristics that are applied during query optimization.

19.5. How does a query tree represent a relational algebra expression? What is meant by an execution of a query tree? Discuss the rules for transformation of query trees and identify when each rule should be applied during optimization.

19.6. How many different join orders are there for a query that joins 10 relations?

19.7. What is meant by *cost-based query optimization*?

19.8. What is the difference between *pipelining* and *materialization*?

19.9. Discuss the cost components for a cost function that is used to estimate query execution cost. Which cost components are used most often as the basis for cost functions?

19.10. Discuss the different types of parameters that are used in cost functions. Where is this information kept?

19.11. List the cost functions for the SELECT and JOIN methods discussed in Section 19.8.

19.12. What is meant by semantic query optimization? How does it differ from other query optimization techniques?

Exercises

19.13. Consider SQL queries Q1, Q8, Q1B, and Q4 in Chapter 4 and Q27 in Chapter 5.

a. Draw at least two query trees that can represent *each* of these queries. Under what circumstances would you use each of your query trees?

b. Draw the initial query tree for each of these queries, and then show how the query tree is optimized by the algorithm outlined in Section 19.7.

c. For each query, compare your own query trees of part (a) and the initial and final query trees of part (b).

19.14. A file of 4096 blocks is to be sorted with an available buffer space of 64 blocks. How many passes will be needed in the merge phase of the external sort-merge algorithm?

19.15. Develop cost functions for the PROJECT, UNION, INTERSECTION, SET DIFFERENCE, and CARTESIAN PRODUCT algorithms discussed in Section 19.4.

19.16. Develop cost functions for an algorithm that consists of two SELECTs, a JOIN, and a final PROJECT, in terms of the cost functions for the individual operations.

19.17. Can a nondense index be used in the implementation of an aggregate operator? Why or why not?

19.18. Calculate the cost functions for different options of executing the JOIN operation OP7 discussed in Section 19.3.2.

19.19. Develop formulas for the hybrid hash-join algorithm for calculating the size of the buffer for the first bucket. Develop more accurate cost estimation formulas for the algorithm.

19.20. Estimate the cost of operations OP6 and OP7, using the formulas developed in Exercise 19.9.

19.21. Extend the sort-merge join algorithm to implement the LEFT OUTER JOIN operation.

19.22. Compare the cost of two different query plans for the following query:

$$\sigma_{\text{Salary} > 40000}(\text{EMPLOYEE} \bowtie_{\text{Dno=Dnumber}} \text{DEPARTMENT})$$

Use the database statistics in Figure 19.8.

Selected Bibliography

A detailed algorithm for relational algebra optimization is given by Smith and Chang (1975). The Ph.D. thesis of Kooi (1980) provides a foundation for query processing techniques. A survey paper by Jarke and Koch (1984) gives a taxonomy of query optimization and includes a bibliography of work in this area. A survey by Graefe (1993) discusses query execution in database systems and includes an extensive bibliography.

Whang (1985) discusses query optimization in OBE (Office-By-Example), which is a system based on the language QBE. Cost-based optimization was introduced in the SYSTEM R experimental DBMS and is discussed in Astrahan et al. (1976). Selinger et al. (1979) is a classic paper that discussed cost-based optimization of multiway joins in SYSTEM R. Join algorithms are discussed in Gotlieb (1975), Blasgen and Eswaran (1976), and Whang et al. (1982). Hashing algorithms for implementing joins are described and analyzed in DeWitt et al. (1984), Bratbergsengen (1984), Shapiro (1986), Kitsuregawa et al. (1989), and Blakeley and Martin (1990), among others. Approaches to finding a good join order are presented in Ioannidis and Kang (1990) and in Swami and Gupta (1989). A discussion of the implications of left-deep and bushy join trees is presented in Ioannidis and Kang (1991). Kim (1982) discusses transformations of nested SQL queries into canonical representations. Optimization of aggregate functions is discussed in Klug (1982) and Muralikrishna (1992). Salzberg et al. (1990) describe a fast external sorting algorithm. Estimating the size of temporary relations is crucial for query optimization. Sampling-based estimation schemes are presented in Haas et al. (1995) and in Haas and Swami (1995). Lipton et al. (1990) also discuss selectivity estimation. Having the database system store and use more detailed statistics in the form of histograms is the topic of Muralikrishna and DeWitt (1988) and Poosala et al. (1996).

Kim et al. (1985) discuss advanced topics in query optimization. Semantic query optimization is discussed in King (1981) and Malley and Zdonick (1986). Work on semantic query optimization is reported in Chakravarthy et al. (1990), Shenoy and Ozsoyoglu (1989), and Siegel et al. (1992).

chapter 20

Physical Database Design and Tuning

In the last chapter we discussed various techniques by which queries can be processed efficiently by the DBMS. These techniques are mostly internal to the DBMS and invisible to the programmer. In this chapter we discuss additional issues that affect the performance of an application running on a DBMS. In particular, we discuss some of the options available to database administrators and programmers for storing databases, and some of the heuristics, rules, and techniques that they can use to tune the database for performance improvement. First, in Section 20.1, we discuss the issues that arise in physical database design dealing with storage and access of data. Then, in Section 20.2, we discuss how to improve database performance through tuning, indexing of data, database design, and the queries themselves.

20.1 Physical Database Design in Relational Databases

In this section, we begin by discussing the physical design factors that affect the performance of applications and transactions, and then we comment on the specific guidelines for RDBMSs.

20.1.1 Factors That Influence Physical Database Design

Physical design is an activity where the goal is not only to create the appropriate structuring of data in storage, but also to do so in a way that guarantees good performance. For a given conceptual schema, there are many physical design alternatives in a given DBMS. It is not possible to make meaningful physical design

decisions and performance analyses until the database designer knows the mix of queries, transactions, and applications that are expected to run on the database. This is called the **job mix** for the particular set of database system applications. The database administrators/designers must analyze these applications, their expected frequencies of invocation, any timing constraints on their execution speed, the expected frequency of update operations, and any unique constraints on attributes. We discuss each of these factors next.

A. Analyzing the Database Queries and Transactions. Before undertaking the physical database design, we must have a good idea of the intended use of the database by defining in a high-level form the queries and transactions that are expected to run on the database. For each **retrieval query**, the following information about the query would be needed:

1. The files that will be accessed by the query.[1]
2. The attributes on which any selection conditions for the query are specified.
3. Whether the selection condition is an equality, inequality, or a range condition.
4. The attributes on which any join conditions or conditions to link multiple tables or objects for the query are specified.
5. The attributes whose values will be retrieved by the query.

The attributes listed in items 2 and 4 above are candidates for the definition of access structures, such as indexes, hash keys, or sorting of the file.

For each **update operation** or **update transaction**, the following information would be needed:

1. The files that will be updated.
2. The type of operation on each file (insert, update, or delete).
3. The attributes on which selection conditions for a delete or update are specified.
4. The attributes whose values will be changed by an update operation.

Again, the attributes listed in item 3 are candidates for access structures on the files, because they would be used to locate the records that will be updated or deleted. On the other hand, the attributes listed in item 4 are candidates for *avoiding an access structure*, since modifying them will require updating the access structures.

B. Analyzing the Expected Frequency of Invocation of Queries and Transactions. Besides identifying the characteristics of expected retrieval queries and update transactions, we must consider their expected rates of invocation. This frequency information, along with the attribute information collected on each query and transaction, is used to compile a cumulative list of the expected frequency of use for all queries and transactions. This is expressed as the expected frequency of using each attribute in each file as a selection attribute or a join attribute,

[1]For simplicity we use the term *files* here, but this can also mean tables or relations.

over all the queries and transactions. Generally, for large volumes of processing, the informal *80–20 rule* can be used: approximately 80 percent of the processing is accounted for by only 20 percent of the queries and transactions. Therefore, in practical situations, it is rarely necessary to collect exhaustive statistics and invocation rates on all the queries and transactions; it is sufficient to determine the 20 percent or so most important ones.

C. Analyzing the Time Constraints of Queries and Transactions. Some queries and transactions may have stringent performance constraints. For example, a transaction may have the constraint that it should terminate within 5 seconds on 95 percent of the occasions when it is invoked, and that it should never take more than 20 seconds. Such timing constraints place further priorities on the attributes that are candidates for access paths. The selection attributes used by queries and transactions with time constraints become higher-priority candidates for primary access structures for the files, because the primary access structures are generally the most efficient for locating records in a file.

D. Analyzing the Expected Frequencies of Update Operations. A minimum number of access paths should be specified for a file that is frequently updated, because updating the access paths themselves slows down the update operations. For example, if a file that has frequent record insertions has 10 indexes on 10 different attributes, each of these indexes must be updated whenever a new record is inserted. The overhead for updating 10 indexes can slow down the insert operations.

E. Analyzing the Uniqueness Constraints on Attributes. Access paths should be specified on all *candidate key* attributes—or sets of attributes—that are either the primary key of a file or unique attributes. The existence of an index (or other access path) makes it sufficient to only search the index when checking this uniqueness constraint, since all values of the attribute will exist in the leaf nodes of the index. For example, when inserting a new record, if a key attribute value of the new record *already exists in the index*, the insertion of the new record should be rejected, since it would violate the uniqueness constraint on the attribute.

Once the preceding information is compiled, it is possible to address the physical database design decisions, which consist mainly of deciding on the storage structures and access paths for the database files.

20.1.2 Physical Database Design Decisions

Most relational systems represent each base relation as a physical database file. The access path options include specifying the type of primary file organization for each relation and the attributes of which indexes that should be defined. At most, one of the indexes on each file may be a primary or a clustering index. Any number of additional secondary indexes can be created.[2]

[2]The reader should review the various types of indexes described in Section 18.1. For a clearer understanding of this discussion, it is also helpful to be familiar with the algorithms for query processing discussed in Chapter 19.

Design Decisions about Indexing. The attributes whose values are required in equality or range conditions (selection operation) are those that are keys or that participate in join conditions (join operation) requiring access paths, such as indexes.

The performance of queries largely depends upon what indexes or hashing schemes exist to expedite the processing of selections and joins. On the other hand, during insert, delete, or update operations, the existence of indexes adds to the overhead. This overhead must be justified in terms of the gain in efficiency by expediting queries and transactions.

The physical design decisions for indexing fall into the following categories:

1. **Whether to index an attribute.** The general rules for creating an index on an attribute are that the attribute must either be a key (unique), or there must be some query that uses that attribute either in a selection condition (equality or range of values) or in a join condition. One reason for creating multiple indexes is that some operations can be processed by just scanning the indexes, without having to access the actual data file (see Section 19.5).
2. **What attribute or attributes to index on.** An index can be constructed on a single attribute, or on more than one attribute if it is a composite index. If multiple attributes from one relation are involved together in several queries, (for example, (Garment_style_#, Color) in a garment inventory database), a multiattribute (composite) index is warranted. The ordering of attributes within a multiattribute index must correspond to the queries. For instance, the above index assumes that queries would be based on an ordering of colors within a Garment_style_# rather than vice versa.
3. **Whether to set up a clustered index.** At most, one index per table can be a primary or clustering index, because this implies that the file be physically ordered on that attribute. In most RDBMSs, this is specified by the keyword CLUSTER. (If the attribute is a *key*, *a primary index* is created, whereas a *clustering index* is created if the attribute is *not a key*—see Section 18.1.) If a table requires several indexes, the decision about which one should be the primary or clustering index depends upon whether keeping the table ordered on that attribute is needed. Range queries benefit a great deal from clustering. If several attributes require range queries, relative benefits must be evaluated before deciding which attribute to cluster on. If a query is to be answered by doing an index search only (without retrieving data records), the corresponding index should *not* be clustered, since the main benefit of clustering is achieved when retrieving the records themselves. A clustering index may be set up as a multiattribute index if range retrieval by that composite key is useful in report creation (for example, an index on Zip_code, Store_id, and Product_id may be a clustering index for sales data).
4. **Whether to use a hash index over a tree index.** In general, RDBMSs use B^+-trees for indexing. However, ISAM and hash indexes are also provided in some systems (see Chapter 18). B^+-trees support both equality and range queries on the attribute used as the search key. Hash indexes work well with

equality conditions, particularly during joins to find a matching record(s), but they do not support range queries.

5. **Whether to use dynamic hashing for the file.** For files that are very volatile—that is, those that grow and shrink continuously—one of the dynamic hashing schemes discussed in Section 17.9 would be suitable. Currently, they are not offered by many commercial RDBMSs.

How to Create an Index. Many RDBMSs have a similar type of command for creating an index, although it is not part of the SQL standard. The general form of this command is:

CREATE [UNIQUE] INDEX <index name>
ON <table name> (<column name> [<order>] { , <column name> [<order>] })
[CLUSTER] ;

The keywords UNIQUE and CLUSTER are optional. The keyword CLUSTER is used when the index to be created should also sort the data file records on the indexing attribute. Thus, specifying CLUSTER on a key (unique) attribute would create some variation of a primary index, whereas specifying CLUSTER on a nonkey (nonunique) attribute would create some variation of a clustering index. The value for <order> can be either ASC (ascending) or DESC (descending), and specifies whether the data file should be ordered in ascending or descending values of the indexing attribute. The default is ASC. For example, the following would create a clustering (ascending) index on the nonkey attribute Dno of the EMPLOYEE file:

CREATE INDEX DnoIndex
ON EMPLOYEE (Dno)
CLUSTER ;

Denormalization as a Design Decision for Speeding Up Queries. The ultimate goal during normalization (see Chapters 15 and 16) is to separate attributes into tables to minimize redundancy, and thereby avoid the update anomalies that lead to an extra processing overhead to maintain consistency in the database. The ideals that are typically followed are the third or Boyce-Codd normal forms (see Chapter 15).

The above ideals are sometimes sacrificed in favor of faster execution of frequently occurring queries and transactions. This process of storing the logical database design (which may be in BCNF or 4NF) in a weaker normal form, say 2NF or 1NF, is called **denormalization**. Typically, the designer includes certain attributes from a table S into another table R. The reason is that the attributes from S that are included in R are frequently needed—along with other attributes in R—for answering queries or producing reports. By including these attributes, a join of R with S is avoided for these frequently occurring queries and reports. This reintroduces *redundancy* in the base tables by including the same attributes in both tables R and S. A partial functional dependency or a transitive dependency now exists in the table R, thereby creating the associated redundancy problems (see Chapter 15). A tradeoff exists between the additional updating needed for maintaining consistency of

redundant attributes versus the effort needed to perform a join to incorporate the additional attributes needed in the result. For example, consider the following relation:

ASSIGN (Emp_id, Proj_id, Emp_name, Emp_job_title, Percent_assigned, Proj_name, Proj_mgr_id, Proj_mgr_name),

which corresponds exactly to the headers in a report called *The Employee Assignment Roster*.

This relation is only in 1NF because of the following functional dependencies:

Proj_id → Proj_name, Proj_mgr_id
Proj_mgr_id → Proj_mgr_name
Emp_id → Emp_name, Emp_job_title

This relation may be preferred over the design in 2NF (and 3NF) consisting of the following three relations:

EMP (Emp_id, Emp_name, Emp_job_title)
PROJ (Proj_id, Proj_name, Proj_mgr_id)
EMP_PROJ (Emp_id, Proj_id, Percent_assigned)

This is because to produce the *The Employee Assignment Roster* report (with all fields shown in ASSIGN above), the latter multirelation design requires two NATURAL JOIN (indicated with *) operations (between EMP and EMP_PROJ, and between PROJ and EMP_PROJ), plus a final JOIN between PROJ and EMP to retrieve the Proj_mgr_name from the Proj_mgr_id. Thus the following JOINs would be needed (the final join would also require renaming (aliasing) of the last EMP table, which is not shown):

$$((\text{EMP\_PROJ} * \text{EMP}) * \text{PROJ}) \bowtie_{\text{PROJ.Proj\_mgr\_id} = \text{EMP.Emp\_id}} \text{EMP}$$

It is also possible to create a view for the ASSIGN table. This does not mean that the join operations will be avoided, but that the user need not specify the joins. If the view table is materialized, the joins would be avoided, but if the virtual view table is not stored as a materialized file, the join computations would still be necessary. Other forms of denormalization consist of storing extra tables to maintain original functional dependencies that are lost during BCNF decomposition. For example, Figure 15.14 shows the TEACH(Student, Course, Instructor) relation with the functional dependencies {{Student, Course} → Instructor, Instructor → Course}. A lossless decomposition of TEACH into T1(Student, Instructor) and T2(Instructor, Course) does *not* allow queries of the form *what course did student Smith take from instructor Navathe* to be answered without joining T1 and T2. Therefore, storing T1, T2, and TEACH may be a possible solution, which reduces the design from BCNF to 3NF. Here, TEACH is a materialized join of the other two tables, representing an extreme redundancy. Any updates to T1 and T2 would have to be applied to TEACH. An alternate strategy is to create T1 and T2 as updatable base tables, and to create TEACH as a view (virtual table) on T1 and T2 that can only be queried.

20.2 An Overview of Database Tuning in Relational Systems

After a database is deployed and is in operation, actual use of the applications, transactions, queries, and views reveals factors and problem areas that may not have been accounted for during the initial physical design. The inputs to physical design listed in Section 20.1.1 can be revised by gathering actual statistics about usage patterns. Resource utilization as well as internal DBMS processing—such as query optimization—can be monitored to reveal bottlenecks, such as contention for the same data or devices. Volumes of activity and sizes of data can be better estimated. Therefore, it is necessary to monitor and revise the physical database design constantly—an activity referred to as **database tuning**. The goals of tuning are as follows:

- To make applications run faster.
- To improve (lower) the response time of queries and transactions.
- To improve the overall throughput of transactions.

The dividing line between physical design and tuning is very thin. The same design decisions that we discussed in Section 20.1.2 are revisited during database tuning, which is a continual adjustment of the physical design. We give a brief overview of the tuning process below.[3] The inputs to the tuning process include statistics related to the same factors mentioned in Section 20.1.1. In particular, DBMSs can internally collect the following statistics:

- Sizes of individual tables.
- Number of distinct values in a column.
- The number of times a particular query or transaction is submitted and executed in an interval of time.
- The times required for different phases of query and transaction processing (for a given set of queries or transactions).

These and other statistics create a profile of the contents and use of the database. Other information obtained from monitoring the database system activities and processes includes the following:

- **Storage statistics.** Data about allocation of storage into tablespaces, indexspaces, and buffer pools.
- **I/O and device performance statistics.** Total read/write activity (paging) on disk extents and disk hot spots.
- **Query/transaction processing statistics.** Execution times of queries and transactions, and optimization times during query optimization.

[3]Interested readers should consult Shasha and Bonnet (2002) for a detailed discussion of tuning.

- **Locking/logging related statistics.** Rates of issuing different types of locks, transaction throughput rates, and log records activity.[4]
- **Index statistics.** Number of levels in an index, number of noncontiguous leaf pages, and so on.

Some of the above statistics relate to transactions, concurrency control, and recovery, which are discussed in Chapters 21 through 23. Tuning a database involves dealing with the following types of problems:

- How to avoid excessive lock contention, thereby increasing concurrency among transactions.
- How to minimize the overhead of logging and unnecessary dumping of data.
- How to optimize the buffer size and scheduling of processes.
- How to allocate resources such as disks, RAM, and processes for most efficient utilization.

Most of the previously mentioned problems can be solved by the DBA by setting appropriate physical DBMS parameters, changing configurations of devices, changing operating system parameters, and other similar activities. The solutions tend to be closely tied to specific systems. The DBAs are typically trained to handle these tuning problems for the specific DBMS. We briefly discuss the tuning of various physical database design decisions below.

20.2.1 Tuning Indexes

The initial choice of indexes may have to be revised for the following reasons:

- Certain queries may take too long to run for lack of an index.
- Certain indexes may not get utilized at all.
- Certain indexes may undergo too much updating because the index is on an attribute that undergoes frequent changes.

Most DBMSs have a command or trace facility, which can be used by the DBA to ask the system to show how a query was executed—what operations were performed in what order and what secondary access structures (indexes) were used. By analyzing these execution plans, it is possible to diagnose the causes of the above problems. Some indexes may be dropped and some new indexes may be created based on the tuning analysis.

The goal of tuning is to dynamically evaluate the requirements, which sometimes fluctuate seasonally or during different times of the month or week, and to reorganize the indexes and file organizations to yield the best overall performance. Dropping and building new indexes is an overhead that can be justified in terms of performance improvements. Updating of a table is generally suspended while an

[4]The reader should preview Chapters 21–23 for an explanation of these terms.

index is dropped or created; this loss of service must be accounted for. Besides dropping or creating indexes and changing from a nonclustered to a clustered index and vice versa, **rebuilding the index** may improve performance. Most RDBMSs use B^+-trees for an index. If there are many deletions on the index key, index pages may contain wasted space, which can be claimed during a rebuild operation. Similarly, too many insertions may cause overflows in a clustered index that affect performance. Rebuilding a clustered index amounts to reorganizing the entire table ordered on that key.

The available options for indexing and the way they are defined, created, and reorganized varies from system to system. As an illustration, consider the sparse and dense indexes in Chapter 18. A sparse index such as a primary index (see Section 18.1) will have one index pointer for each page (disk block) in the data file; a dense index such as a unique secondary index will have an index pointer for each record. Sybase provides clustering indexes as sparse indexes in the form of B^+-trees, whereas INGRES provides sparse clustering indexes as ISAM files and dense clustering indexes as B^+-trees. In some versions of Oracle and DB2, the option of setting up a clustering index is limited to a dense index (with many more index entries), and the DBA has to work with this limitation.

20.2.2 Tuning the Database Design

In Section 20.1.2, we discussed the need for a possible denormalization, which is a departure from keeping all tables as BCNF relations. If a given physical database design does not meet the expected objectives, the DBA may revert to the logical database design, make adjustments such as denormalizations to the logical schema, and remap it to a new set of physical tables and indexes.

As discussed, the entire database design has to be driven by the processing requirements as much as by data requirements. If the processing requirements are dynamically changing, the design needs to respond by making changes to the conceptual schema if necessary and to reflect those changes into the logical schema and physical design. These changes may be of the following nature:

- Existing tables may be joined (denormalized) because certain attributes from two or more tables are frequently needed together: This reduces the normalization level from BCNF to 3NF, 2NF, or 1NF.[5]
- For the given set of tables, there may be alternative design choices, all of which achieve 3NF or BCNF. We illustrated alternative equivalent designs in Chapter 16. One normalized design may be replaced by another.
- A relation of the form R(K,A, B, C, D, ...)—with K as a set of key attributes—that is in BCNF can be stored in multiple tables that are also in BCNF—for example, R1(K, A, B), R2(K, C, D,), R3(K, ...)—by replicating the key K in each table. Such a process is known as **vertical partitioning**. Each table groups

[5]Note that 3NF and 2NF address different types of problem dependencies that are independent of each other; hence, the normalization (or denormalization) order between them is arbitrary.

sets of attributes that are accessed together. For example, the table EMPLOYEE(Ssn, Name, Phone, Grade, Salary) may be split into two tables: EMP1(Ssn, Name, Phone) and EMP2(Ssn, Grade, Salary). If the original table has a large number of rows (say 100,000) and queries about phone numbers and salary information are totally distinct and occur with very different frequencies, then this separation of tables may work better.

- Attribute(s) from one table may be repeated in another even though this creates redundancy and a potential anomaly. For example, Part_name may be replicated in tables wherever the Part# appears (as foreign key), but there may be one master table called PART_MASTER(Part#, Part_name, ...) where the Partname is guaranteed to be up-to-date.
- Just as vertical partitioning splits a table vertically into multiple tables, **horizontal partitioning** takes horizontal slices of a table and stores them as distinct tables. For example, product sales data may be separated into ten tables based on ten product lines. Each table has the same set of columns (attributes) but contains a distinct set of products (tuples). If a query or transaction applies to all product data, it may have to run against all the tables and the results may have to be combined.

These types of adjustments designed to meet the high volume of queries or transactions, with or without sacrificing the normal forms, are commonplace in practice.

20.2.3 Tuning Queries

We already discussed how query performance is dependent upon the appropriate selection of indexes, and how indexes may have to be tuned after analyzing queries that give poor performance by using the commands in the RDBMS that show the execution plan of the query. There are mainly two indications that suggest that query tuning may be needed:

1. A query issues too many disk accesses (for example, an exact match query scans an entire table).
2. The query plan shows that relevant indexes are not being used.

Some typical instances of situations prompting query tuning include the following:

1. Many query optimizers do not use indexes in the presence of arithmetic expressions (such as Salary/365 > 10.50), numerical comparisons of attributes of different sizes and precision (such as Aqty = Bqty where Aqty is of type INTEGER and Bqty is of type SMALLINTEGER), NULL comparisons (such as Bdate IS NULL), and substring comparisons (such as Lname LIKE '%mann').
2. Indexes are often not used for nested queries using IN; for example, the following query:

```
SELECT  Ssn     FROM   EMPLOYEE
WHERE   Dno     IN (   SELECT Dnumber FROM DEPARTMENT
                       WHERE Mgr_ssn = '333445555' );
```

may not use the index on Dno in EMPLOYEE, whereas using Dno = Dnumber in the WHERE-clause with a single block query may cause the index to be used.

3. Some DISTINCTs may be redundant and can be avoided without changing the result. A DISTINCT often causes a sort operation and must be avoided as much as possible.
4. Unnecessary use of temporary result tables can be avoided by collapsing multiple queries into a single query *unless* the temporary relation is needed for some intermediate processing.
5. In some situations involving the use of correlated queries, temporaries are useful. Consider the following query, which retrieves the highest paid employee in each department:

```
SELECT   Ssn
FROM     EMPLOYEE E
WHERE    Salary = SELECT MAX (Salary)
                  FROM EMPLOYEE AS M
                  WHERE M.Dno = E.Dno;
```

   This has the potential danger of searching all of the inner EMPLOYEE table M for *each* tuple from the outer EMPLOYEE table E. To make the execution more efficient, the process can be broken into two queries, where the first query just computes the maximum salary in each department as follows:

```
SELECT   MAX (Salary) AS High_salary, Dno INTO TEMP
FROM     EMPLOYEE
GROUP BY Dno;
SELECT   EMPLOYEE.Ssn
FROM     EMPLOYEE, TEMP
WHERE    EMPLOYEE.Salary = TEMP.High_salary
         AND EMPLOYEE.Dno = TEMP.Dno;
```

6. If multiple options for a join condition are possible, choose one that uses a clustering index and avoid those that contain string comparisons. For example, assuming that the Name attribute is a candidate key in EMPLOYEE and STUDENT, it is better to use EMPLOYEE.Ssn = STUDENT.Ssn as a join condition rather than EMPLOYEE.Name = STUDENT.Name if Ssn has a clustering index in one or both tables.
7. One idiosyncrasy with some query optimizers is that the order of tables in the FROM-clause may affect the join processing. If that is the case, one may have to switch this order so that the smaller of the two relations is scanned and the larger relation is used with an appropriate index.
8. Some query optimizers perform worse on nested queries compared to their equivalent unnested counterparts. There are four types of nested queries:
   - Uncorrelated subqueries with aggregates in an inner query.
   - Uncorrelated subqueries without aggregates.
   - Correlated subqueries with aggregates in an inner query.

- Correlated subqueries without aggregates.

Of the four types above, the first one typically presents no problem, since most query optimizers evaluate the inner query once. However, for a query of the second type, such as the example in item 2, most query optimizers may not use an index on Dno in EMPLOYEE. However, the same optimizers may do so if the query is written as an unnested query. Transformation of correlated subqueries may involve setting temporary tables. Detailed examples are outside our scope here.[6]

9. Finally, many applications are based on views that define the data of interest to those applications. Sometimes, these views become overkill, because a query may be posed directly against a base table, rather than going through a view that is defined by a JOIN.

20.2.4 Additional Query Tuning Guidelines

Additional techniques for improving queries apply in certain situations as follows:

1. A query with multiple selection conditions that are connected via OR may not be prompting the query optimizer to use any index. Such a query may be split up and expressed as a union of queries, each with a condition on an attribute that causes an index to be used. For example,

```
SELECT  Fname, Lname, Salary, Age[7]
FROM    EMPLOYEE
WHERE   Age > 45 OR Salary < 50000;
```

may be executed using sequential scan giving poor performance. Splitting it up as

```
SELECT  Fname, Lname, Salary, Age
FROM    EMPLOYEE
WHERE   Age > 45
UNION
SELECT  Fname, Lname, Salary, Age
FROM    EMPLOYEE
WHERE   Salary < 50000;
```

may utilize indexes on Age as well as on Salary.

2. To help expedite a query, the following transformations may be tried:
   - NOT condition may be transformed into a positive expression.
   - Embedded SELECT blocks using IN, = ALL, = ANY, and = SOME may be replaced by joins.
   - If an equality join is set up between two tables, the range predicate (selection condition) on the joining attribute set up in one table may be repeated for the other table.

[6]For further details, see Shasha and Bonnet (2002).

[7]We modified the schema and used Age in EMPLOYEE instead of Bdate.

3. WHERE conditions may be rewritten to utilize the indexes on multiple columns. For example,

```
SELECT  Region#, Prod_type, Month, Sales
FROM    SALES_STATISTICS
WHERE   Region# = 3 AND ((Prod_type BETWEEN 1 AND 3) OR (Prod_type
        BETWEEN 8 AND 10));
```

may use an index only on Region# and search through all leaf pages of the index for a match on Prod_type. Instead, using

```
SELECT  Region#, Prod_type, Month, Sales
FROM    SALES_STATISTICS
WHERE   (Region# = 3 AND (Prod_type BETWEEN 1 AND 3))
        OR (Region# = 3 AND (Prod_type BETWEEN 8 AND 10));
```

may use a composite index on (Region#, Prod_type) and work much more efficiently.

In this section, we have covered many of the common instances where the inefficiency of a query may be fixed by some simple corrective action such as using a temporary table, avoiding certain types of query constructs, or avoiding the use of views. The goal is to have the RDBMS use existing single attribute or composite attribute indexes as much as possible. This avoids full scans of data blocks or entire scanning of index leaf nodes. Redundant processes like sorting must be avoided at any cost. The problems and the remedies will depend upon the workings of a query optimizer within an RDBMS. Detailed literature exists in database tuning guidelines for database administration by the RDBMS vendors. Major relational DBMS vendors like Oracle, IBM and Microsoft encourage their large customers to share ideas of tuning at the annual expos and other forums so that the entire industry benefits by using performance enhancement techniques. These techniques are typically available in trade literature and on various Web sites.

20.3 Summary

In this chapter, we discussed the factors that affect physical database design decisions and provided guidelines for choosing among physical design alternatives. We discussed changes to logical design such as denormalization, as well as modifications of indexes, and changes to queries to illustrate different techniques for database performance tuning. These are only a representative sample of a large number of measures and techniques adopted in the design of large commercial applications of relational DBMSs.

Review Questions

20.1. What are the important factors that influence physical database design?

20.2. Discuss the decisions made during physical database design.

20.3. Discuss the guidelines for physical database design in RDBMSs.

20.4. Discuss the types of modifications that may be applied to the logical database design of a relational database.

20.5. Under what situations would denormalization of a database schema be used? Give examples of denormalization.

20.6. Discuss the tuning of indexes for relational databases.

20.7. Discuss the considerations for reevaluating and modifying SQL queries.

20.8. Illustrate the types of changes to SQL queries that may be worth considering for improving the performance during database tuning.

Selected Bibliography

Wiederhold (1987) covers issues related to physical design. O'Neil and O'Neil (2001) has a detailed discussion of physical design and transaction issues in reference to commercial RDBMSs. Navathe and Kerschberg (1986) discuss all phases of database design and point out the role of data dictionaries. Rozen and Shasha (1991) and Carlis and March (1984) present different models for the problem of physical database design. Shasha and Bonnet (2002) has an elaborate discussion of guidelines for database tuning. Niemiec (2008) is one among several books available for Oracle database administration and tuning; Schneider (2006) is focused on designing and tuning MySQL databases.

part 9

Transaction Processing, Concurrency Control, and Recovery

chapter 21

Introduction to Transaction Processing Concepts and Theory

The concept of transaction provides a mechanism for describing logical units of database processing. **Transaction processing systems** are systems with large databases and hundreds of concurrent users executing database transactions. Examples of such systems include airline reservations, banking, credit card processing, online retail purchasing, stock markets, supermarket checkouts, and many other applications. These systems require high availability and fast response time for hundreds of concurrent users. In this chapter we present the concepts that are needed in transaction processing systems. We define the concept of a transaction, which is used to represent a logical unit of database processing that must be completed in its entirety to ensure correctness. A transaction is typically implemented by a computer program, which includes database commands such as retrievals, insertions, deletions, and updates. We introduced some of the basic techniques for database programming in Chapters 13 and 14.

In this chapter, we focus on the basic concepts and theory that are needed to ensure the correct executions of transactions. 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 the problems that can occur when transactions fail, and how the database system can recover from various types of failures.

This chapter is organized as follows. Section 21.1 informally discusses why concurrency control and recovery are necessary in a database system. Section 21.2 defines the term *transaction* and discusses additional concepts related to transaction processing in database systems. Section 21.3 presents the important properties of atomicity, consistency preservation, isolation, and durability or permanency—called the

ACID properties—that are considered desirable in transaction processing systems. Section 21.4 introduces the concept of schedules (or histories) of executing transactions and characterizes the *recoverability* of schedules. Section 21.5 discusses the notion of *serializability* of concurrent transaction execution, which can be used to define correct execution sequences (or schedules) of concurrent transactions. In Section 21.6, we present some of the commands that support the transaction concept in SQL. Section 21.7 summarizes the chapter.

The two following chapters continue with more details on the actual methods and techniques used to support transaction processing. Chapter 22 gives an overview of the basic concurrency control protocols and Chapter 23 introduces recovery techniques.

21.1 Introduction to Transaction Processing

In this section we discuss the concepts of concurrent execution of transactions and recovery from transaction failures. Section 21.1.1 compares single-user and multiuser database systems and demonstrates how concurrent execution of transactions can take place in multiuser systems. Section 21.1.2 defines the concept of transaction and presents a simple model of transaction execution based on read and write database operations. This model is used as the basis for defining and formalizing concurrency control and recovery concepts. Section 21.1.3 uses informal examples to show why concurrency control techniques are needed in multiuser systems. Finally, Section 21.1.4 discusses why techniques are needed to handle recovery from system and transaction failures by discussing the different ways in which transactions can fail while executing.

21.1.1 Single-User versus Multiuser Systems

One criterion for classifying a database system is according to the number of users who can use the system **concurrently**. 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—and hence access the database—concurrently. Single-user DBMSs are mostly restricted to personal computer systems; most other DBMSs are multiuser. For example, an airline reservations system is used by hundreds of travel agents and reservation clerks concurrently. Database systems used in banks, insurance agencies, stock exchanges, supermarkets, and many other applications are multiuser systems. In these systems, hundreds or thousands of users are typically operating on the database by submitting transactions concurrently to the system.

Multiple users can access databases—and use computer systems—simultaneously because of the concept of **multiprogramming**, which allows the operating system of the computer to execute multiple programs—or **processes**—at the same time. A single central processing unit (CPU) can only execute at most one process at a time. However, **multiprogramming operating systems** execute some commands from one process, then suspend that process and execute some commands from the next

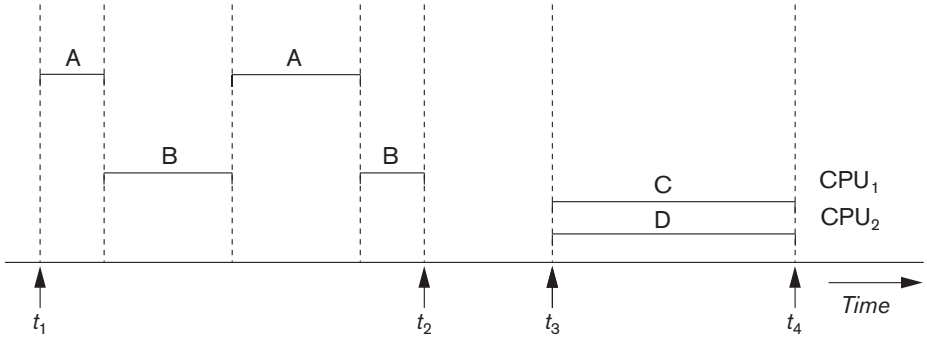


Figure 21.1
Interleaved processing versus parallel processing of concurrent transactions.

process, and so on. A process is resumed at the point where it was suspended whenever it gets its turn to use the CPU again. Hence, concurrent execution of processes is actually **interleaved**, as illustrated in Figure 21.1, which shows two processes, A and B, executing concurrently in an interleaved fashion. Interleaving keeps the CPU busy when a process requires an input or output (I/O) operation, such as reading a block from disk. The CPU is switched to execute another process rather than remaining idle during I/O time. Interleaving also prevents a long process from delaying other processes.

If the computer system has multiple hardware processors (CPUs), **parallel processing** of multiple processes is possible, as illustrated by processes C and D in Figure 21.1. Most of the theory concerning concurrency control in databases is developed in terms of **interleaved concurrency**, so for the remainder of this chapter we assume this model. In a multiuser DBMS, the stored data items are the primary resources that may be accessed concurrently by interactive users or application programs, which are constantly retrieving information from and modifying the database.

21.1.2 Transactions, Database Items, Read and Write Operations, and DBMS Buffers

A **transaction** is an executing program that forms a logical unit of database processing. A transaction includes one or more database access operations—these can include insertion, deletion, modification, or retrieval operations. The database operations that form a transaction can either be embedded within an application program or they can be specified interactively via a high-level query language such as SQL. One way of specifying the transaction boundaries is by specifying explicit **begin transaction** and **end transaction** statements in an application program; in this case, all database access operations between the two are considered as forming one transaction. A single application program may contain more than one transaction if it contains several transaction boundaries. If the database operations in a transaction do not update the database but only retrieve data, the transaction is called a **read-only transaction**; otherwise it is known as a **read-write transaction**.

The *database model* that is used to present transaction processing concepts is quite simple when compared to the data models that we discussed earlier in the book, such as the relational model or the object model. A **database** is basically represented as a collection of *named data items.* The size of a data item is called its **granularity**. A **data item** can be a *database record*, but it can also be a larger unit such as a whole *disk block*, or even a smaller unit such as an individual *field (attribute) value* of some record in the database. The transaction processing concepts we discuss are independent of the data item granularity (size) and apply to data items in general. Each data item has a *unique name*, but this name is not typically used by the programmer; rather, it is just a means to *uniquely identify each data item*. For example, if the data item granularity is one disk block, then the disk block address can be used as the data item name. Using this simplified database model, the basic database access operations that a transaction can include are as follows:

- **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 17, the basic unit of data transfer from disk to main memory is one 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).

It is step 4 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 a modified disk block whose contents are in a main memory buffer is handled by the recovery manager of the DBMS in cooperation with the underlying operating system. The DBMS will maintain in the **database cache** a number of **data buffers** in main memory. Each buffer typically holds the contents of one database disk block, which contains some of the database items being processed. When these buffers are all occupied, and additional database disk blocks must be copied into memory, some buffer replacement policy is used to

choose which of the current buffers is to be replaced. If the chosen buffer has been modified, it must be written back to disk before it is reused.[1]

A transaction includes read_item and write_item operations to access and update the database. Figure 21.2 shows examples of two very simple transactions. The **read-set** of a transaction is the set of all items that the transaction reads, and the **write-set** is the set of all items that the transaction writes. For example, the read-set of T_1 in Figure 21.2 is {X, Y} and its write-set is also {X, Y}.

Concurrency control and recovery mechanisms are mainly concerned with the database 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 some of the problems that may occur.

21.1.3 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 much simplified airline reservations 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 (uniquely identifiable) data item*, among other information. Figure 21.2(a) shows a transaction T_1 that *transfers N* reservations from one flight whose number of reserved seats is stored in the database item named X to another flight whose number of reserved seats is stored in the database item named Y. Figure 21.2(b) shows a simpler transaction T_2 that just *reserves M* seats on the first flight (X) referenced in transaction T_1.[2] 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.

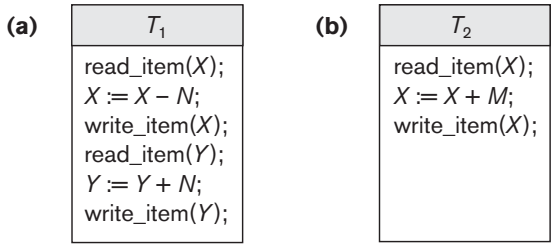


Figure 21.2
Two sample transactions. (a) Transaction T_1. (b) Transaction T_2.

[1]We will not discuss buffer replacement policies here because they are typically discussed in operating systems textbooks.

[2]A similar, more commonly used example assumes a bank database, with one transaction doing a transfer of funds from account X to account Y and the other transaction doing a deposit to account X.

When a database access program is written, it has the flight number, flight date, and the number of seats to be booked as parameters; hence, the same program can be used to execute *many different transactions*, each with a different flight number, date, and number 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 21.2(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. Next we discuss the types of problems we may encounter with these two simple transactions if they run concurrently.

The Lost Update Problem. This problem occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database items incorrect. Suppose that transactions T_1 and T_2 are submitted at approximately the same time, and suppose that their operations are interleaved as shown in Figure 21.3(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 transfers 5 seat reservations from the flight corresponding to X to the flight corresponding to Y), and $M = 4$ (T_2 reserves 4 seats on X), the final result should be $X = 79$. However, in the interleaving of operations shown in Figure 21.3(a), it is $X = 84$ because the update in T_1 that removed the five seats from X was *lost*.

The Temporary Update (or Dirty Read) Problem. This problem occurs when one transaction updates a database item and then the transaction fails for some reason (see Section 21.1.4). Meanwhile, the updated item is accessed (read) by another transaction before it is changed back to its original value. Figure 21.3(b) shows an example where T_1 updates item X and then 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 database items while other transactions are updating some of these items, 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 21.3(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.

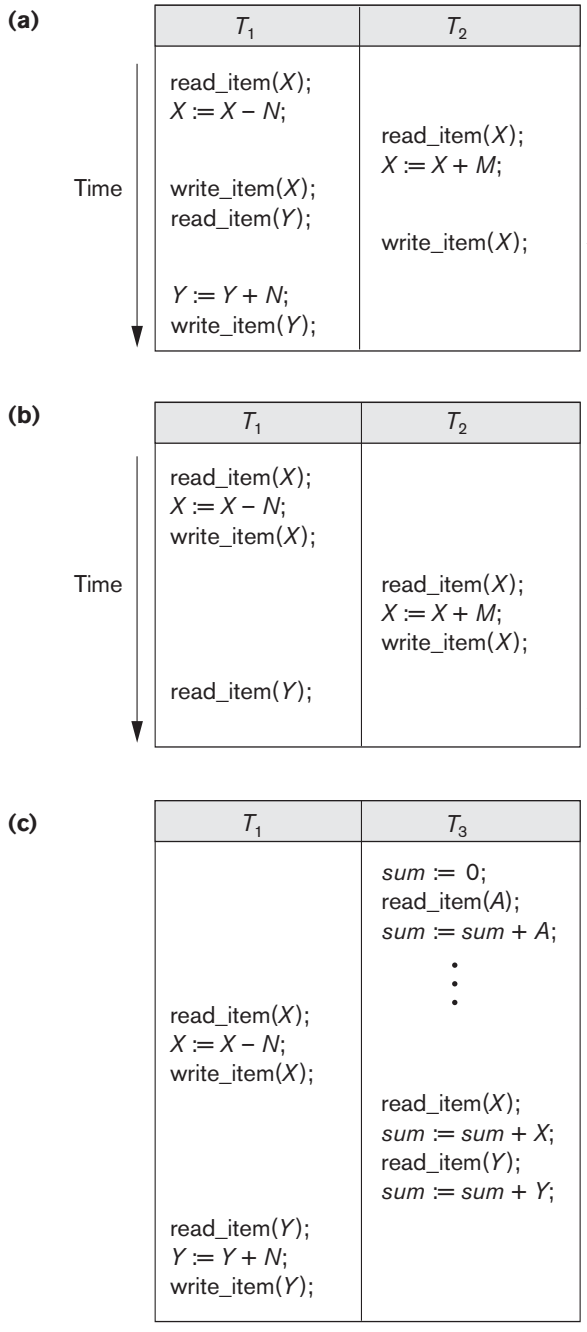


Figure 21.3
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.

The Unrepeatable Read Problem. Another problem that may occur is called *unrepeatable read*, where a transaction T reads the same 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. This may occur, for example, if during an airline reservation transaction, a customer inquires about seat availability on several flights. When the customer decides on a particular flight, the transaction then reads the number of seats on that flight a second time before completing the reservation, and it may end up reading a different value for the item.

21.1.4 Why Recovery Is Needed

Whenever a transaction is submitted to a DBMS for execution, the system is responsible for making sure that either all the operations in the transaction are completed successfully and their effect is recorded permanently in the database, or that the transaction does not have any effect on the database or any other transactions. In the first case, the transaction is said to be **committed**, whereas in the second case, the transaction is **aborted**. The DBMS must not permit some operations of a transaction T to be applied to the database while other operations of T are not, because *the whole transaction* is a logical unit of database processing. If a transaction **fails** after executing some of its operations but before executing all of them, the operations already executed must be undone and have no lasting effect.

Types of Failures. Failures are generally classified as transaction, system, and media failures. There are several possible reasons for a transaction to fail in the middle of execution:

1. **A computer failure (system crash).** A hardware, software, or network error occurs in the computer system during transaction execution. Hardware crashes are usually media failures—for example, main memory failure.
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.[3] Additionally, the user may interrupt the transaction during its execution.
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. An exception condition,[4] such as insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal, to be canceled. This exception could be programmed in the transaction itself, and in such a case would not be considered as a transaction failure.

---

[3]In general, a transaction should be thoroughly tested to ensure that it does not have any bugs (logical programming errors).

[4]Exception conditions, if programmed correctly, do not constitute transaction failures.

4. **Concurrency control enforcement.** The concurrency control method (see Chapter 22) may decide to abort a transaction because it violates serializability (see Section 21.5), or it may abort one or more transactions to resolve a state of deadlock among several transactions (see Section 22.1.3). Transactions aborted because of serializability violations or deadlocks are typically restarted automatically at a later time.
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 quickly recover from the failure. Disk failure or other catastrophic failures of type 5 or 6 do not happen frequently; if they do occur, recovery is a major task. We discuss recovery from failure in Chapter 23.

The concept of transaction is fundamental to many techniques for concurrency control and recovery from failures.

21.2 Transaction and System Concepts

In this section we discuss additional concepts relevant to transaction processing. Section 21.2.1 describes the various states a transaction can be in, and discusses other operations needed in transaction processing. Section 21.2.2 discusses the system log, which keeps information about transactions and data items that will be needed for recovery. Section 21.2.3 describes the concept of commit points of transactions, and why they are important in transaction processing.

21.2.1 Transaction States and Additional Operations

A transaction is an atomic unit of work that should either be completed in its entirety or not done at all. For recovery purposes, the system needs to keep track of when each transaction starts, terminates, and commits or aborts (see Section 21.2.3). Therefore, the recovery manager of the DBMS needs to keep 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 of transaction execution. However, at this point it may be necessary to check whether the changes introduced by

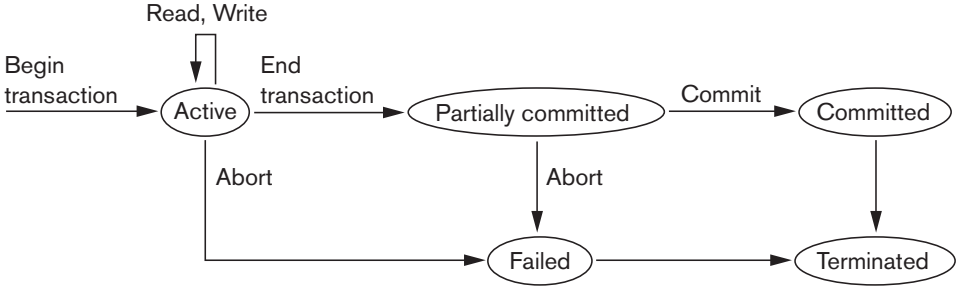
the transaction can be permanently applied to the database (committed) or whether the transaction has to be aborted because it violates serializability (see Section 21.5) or for some other reason.

- 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**.

Figure 21.4 shows a state transition diagram that illustrates how a transaction moves through its execution states. A transaction goes into an **active state** immediately after it starts execution, where it can execute its READ and WRITE operations. When the transaction ends, it moves to the **partially committed state**. At this point, 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 the system log, discussed in the next section).[5] Once this check is 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 21.2.3. When a transaction is committed, it has concluded its execution successfully and all its changes must be recorded permanently in the database, even if a system failure occurs.

However, a transaction can go to the **failed state** if one of the checks fails or if the transaction 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. The transaction information that is maintained in system tables while the transaction has been running is removed when the transaction terminates. Failed or aborted transactions may be *restarted* later—either automatically or after being resubmitted by the user—as brand new transactions.

Figure 21.4
State transition diagram illustrating the states for transaction execution.



[5]Optimistic concurrency control (see Section 22.4) also requires that certain checks are made at this point to ensure that the transaction did not interfere with other executing transactions.

21.2.2 The System Log

To be able to recover from failures that affect transactions, the system maintains a **log**[6] to keep track of all transaction operations that affect the values of database items, as well as other transaction information that may be needed to permit recovery from failures. The log is a sequential, append-only file that is kept on disk, so it is not affected by any type of failure except for disk or catastrophic failure. Typically, one (or more) main memory buffers hold the last part of the log file, so that log entries are first added to the main memory buffer. When the **log buffer** is filled, or when certain other conditions occur, the log buffer is *appended to the end of the log file on disk*. In addition, the log file from disk is periodically backed up to archival storage (tape) to guard against catastrophic failures. The following are the types of entries—called **log records**—that are written to the log file and the corresponding action for each log record. In these entries, T refers to a unique **transaction-id** that is generated automatically by the system for each transaction and that is used to identify each transaction:

1. [**start_transaction, *T*]**. Indicates that transaction T has started execution.
2. [**write_item, *T, X, old_value, new_value***]. Indicates that transaction T has changed the value of database item X from *old_value* to *new_value.*
3. [**read_item, *T, X***]. Indicates that transaction T has read the value of database item X.
4. [**commit, *T***]. Indicates that transaction T has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.
5. [**abort, *T***]. Indicates that transaction T has been aborted.

Protocols for recovery that avoid cascading rollbacks (see Section 21.4.2)—which include nearly all practical protocols—*do not require* that READ operations are written to the system log. However, if the log is also used for other purposes—such as auditing (keeping track of all database operations)—then such entries can be included. Additionally, some recovery protocols require simpler WRITE entries only include one of new_value and old_value instead of including both (see Section 21.4.2).

Notice that we are assuming 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 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 23. 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. **Redo** of an operation may also be necessary if a transaction has its updates recorded in the log but a failure occurs before the system can be sure that

[6]The log has sometimes been called the *DBMS journal.*

all these new_values have been written to the actual database on disk from the main memory buffers.[7]

21.2.3 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 have been recorded in the log. Beyond the commit point, the transaction is said to be **committed**, and its effect must be *permanently recorded* in the database. The transaction then writes a commit record [commit, T] into the log. If a system failure occurs, we can search back in the log for all transactions T that have written a [start_transaction, T] record into the log but have not written their [commit, T] record 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 record in the log must also have recorded all their WRITE operations in the log, so their effect on the database can be *redone* from the log records.

Notice that the log file must be kept on disk. As discussed in Chapter 17, 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 to disk. It is common to keep one or more blocks of the log file in main memory buffers, called the **log buffer**, until they are filled with log entries and then to write them back to disk only once, rather than writing to disk every time a log entry is added. This saves the overhead of multiple disk writes of the same log file buffer. 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 buffer before committing a transaction.

21.3 Desirable Properties of Transactions

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

- **Atomicity.** A transaction is an atomic unit of processing; it should either be performed in its entirety or not performed at all.
- **Consistency preservation.** A transaction should be consistency preserving, meaning that if it is completely executed from beginning to end without interference from other transactions, it should take the database from one consistent state to another.
- **Isolation.** A transaction should appear as though it is being executed in isolation from other transactions, even though many transactions are executing

[7]Undo and redo are discussed more fully in Chapter 23.

concurrently. That is, the execution of a transaction should not be interfered with by any other transactions executing concurrently.

- **Durability or permanency.** The changes applied to the database by a committed transaction must persist in the database. These changes must not be lost because of any failure.

The *atomicity property* requires that we execute a transaction to completion. It is the responsibility of the *transaction recovery subsystem* of a DBMS 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 technique must undo any effects of the transaction on the database. On the other hand, write operations of a committed transaction must be eventually written to disk.

The preservation of *consistency* 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 on the database that should hold. 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.

The *isolation property* is enforced by the *concurrency control subsystem* of the DBMS.[8] If every transaction does not make its updates (write operations) visible to other transactions until it is committed, one form of isolation is enforced that solves the temporary update problem and eliminates cascading rollbacks (see Chapter 23) but does not eliminate all other problems. There have been attempts to define the **level of isolation** of a transaction. A transaction is said to have level 0 (zero) isolation if it does not overwrite the dirty reads of higher-level transactions. Level 1 (one) isolation has no lost updates, and level 2 isolation has no lost updates and no dirty reads. Finally, level 3 isolation (also called *true isolation*) has, in addition to level 2 properties, repeatable reads.[9]

And last, the *durability property* is the responsibility of the *recovery subsystem* of the DBMS. We will introduce how recovery protocols enforce durability and atomicity in the next section and then discuss this in more detail in Chapter 23.

21.4 Characterizing Schedules Based on Recoverability

When transactions are executing concurrently in an interleaved fashion, then the order of execution of operations from all the various transactions is known as a **schedule** (or **history**). In this section, first we define the concept of schedules, and

[8]We will discuss concurrency control protocols in Chapter 22.

[9]The SQL syntax for isolation level discussed later in Section 21.6 is closely related to these levels.

then we characterize the types of schedules that facilitate recovery when failures occur. In Section 21.5, we characterize schedules in terms of the interference of participating transactions, leading to the concepts of serializability and serializable schedules.

21.4.1 Schedules (Histories) of Transactions

A **schedule** (or **history**) S of n transactions $T_1, T_2, ..., T_n$ is an ordering of the operations of the transactions. Operations from different transactions can be interleaved in the schedule S. However, for each transaction T_i that participates in the schedule S, the operations of T_i in S must appear in the same order in which they occur in T_i. The order of operations in S is considered to be a *total ordering*, meaning *that for any two operations* in the schedule, one must occur before the other. It is possible theoretically to deal with schedules whose operations form *partial orders* (as we discuss later), but we will assume for now total ordering of the operations in a schedule.

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 *b*, *r*, *w*, *e*, *c*, and *a* for the operations begin_transaction, read_item, write_item, end_transaction, commit, and abort, respectively, and appends as a *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. In some schedules, we will only show the *read* and *write* operations, whereas in other schedules, we will show all the operations. For example, the schedule in Figure 21.3(a), which we shall call S_a, can be written as follows in this notation:

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

Similarly, the schedule for Figure 21.3(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: $r_1(X)$; $w_1(X)$; $r_2(X)$; $w_2(X)$; $r_1(Y)$; a_1;

Two operations in a schedule are said to **conflict** if they satisfy all three of the following conditions: (1) they belong to *different transactions*; (2) they access the *same item* X; and (3) *at least one* of the 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; the operations $w_2(X)$ and $w_1(Y)$ do not conflict because they operate on distinct data items X and Y; and the operations $r_1(X)$ and $w_1(X)$ do not conflict because they belong to the same transaction.

Intuitively, two operations are conflicting if changing their order can result in a different outcome. For example, if we change the order of the two operations $r_1(X)$; $w_2(X)$ to $w_2(X)$; $r_1(X)$, then the value of X that is read by transaction T_1 changes, because in the second order the value of X is changed by $w_2(X)$ before it is read by

$r_1(X)$, whereas in the first order the value is read before it is changed. This is called a **read-write conflict**. The other type is called a **write-write conflict**, and is illustrated by the case where we change the order of two operations such as $w_1(X)$; $w_2(X)$ to $w_2(X)$; $w_1(X)$. For a write-write conflict, the *last value* of X will differ because in one case it is written by T_2 and in the other case by T_1. Notice that two read operations are not conflicting because changing their order makes no difference in outcome.

The rest of this section covers some theoretical definitions concerning schedules. A schedule S of n transactions T_1, T_2, ..., 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, ..., 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 relative 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.[10]

The preceding condition (3) allows 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.[11] 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* at the end of the schedule.

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.

21.4.2 Characterizing Schedules Based on Recoverability

For some schedules it is easy to recover from transaction and system failures, whereas for other schedules the recovery process can be quite involved. In some cases, it is even not possible to recover correctly after a failure. 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*. These characterizations do not actually provide the recovery algorithm; they only attempt to theoretically characterize the different types of schedules.

[10]Theoretically, it is not necessary to determine an order between pairs of *nonconflicting* operations.

[11]In practice, most schedules have a total order of operations. If parallel processing is employed, it is theoretically possible to have schedules with partially ordered nonconflicting operations.

First, we would like to ensure that, once a transaction T is committed, it should *never* be necessary to roll back T. This ensures that the durability property of transactions is not violated (see Section 21.3). The schedules that theoretically meet this criterion are called *recoverable schedules;* those that do not are called **nonrecoverable** and hence should not be permitted by the DBMS. The definition of **recoverable schedule** is as follows: A schedule S is recoverable if no transaction T in S commits until all transactions T' that have written some item X that T reads have committed. A transaction T **reads** from transaction T' in a schedule S if some item X is first written by T' and later read by T. In addition, T' should not have been 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).

Some recoverable schedules may require a complex recovery process as we shall see, but if sufficient information is kept (in the log), a recovery algorithm can be devised for any recoverable schedule. The (partial) schedules S_a and S_b from the preceding section are both recoverable, since they satisfy the above definition. Consider the schedule S_a' given below, which is the same as schedule S_a except that two commit operations have been added to S_a:

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

S_a' is recoverable, even though it suffers from the lost update problem; this problem is handled by serializability theory (see Section 21.5). 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_e: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$; a_1; a_2;

S_c is not recoverable because T_2 reads item X from T_1, but T_2 commits before T_1 commits. The problem occurs 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 is 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. If T_1 aborts instead of committing, then T_2 should also abort as shown in S_e, because the value of X it read is no longer valid. In S_e, aborting T_2 is acceptable since it has not committed yet, which is not the case for the nonrecoverable schedule S_c.

In a recoverable schedule, no committed transaction ever needs to be rolled back, and so the definition of committed transaction as durable is not violated. However, it is possible for a phenomenon known as **cascading rollback** (or **cascading abort**) to occur in some recoverable schedules, 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.

Because cascading rollback can be quite time-consuming—since numerous transactions can be rolled back (see Chapter 23)—it is important to characterize the sched-

ules where this phenomenon is guaranteed not to occur. A schedule is said to be **cascadeless**, or to **avoid cascading rollback**, if every transaction in the schedule reads only items that were written by committed transactions. In this case, all items read will not be discarded, so no cascading rollback will occur. To satisfy this criterion, the $r_2(X)$ command in schedules S_d and 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 recovery process. In a strict schedule, the process of undoing a write_item(X) operation of an aborted transaction is simply to restore the **before image** (old_value or BFIM) of data item X. This simple procedure always works correctly for strict schedules, but it may not work for recoverable or cascadeless schedules. 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_f, 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 schedule S_f is cascadeless, it is not a strict schedule, since it permits T_2 to write item X even though the transaction T_1 that last wrote X had not yet committed (or aborted). A strict schedule does not have this problem.

It is important to note that any strict schedule is also cascadeless, and any cascadeless schedule is also recoverable. Suppose we have i transactions T_1, T_2, ..., T_i, and their number of operations are n_1, n_2, ..., n_i, respectively. If we make a set of all possible schedules of these transactions, we can divide the schedules into two disjoint subsets: recoverable and nonrecoverable. The cascadeless schedules will be a subset of the recoverable schedules, and the strict schedules will be a subset of the cascadeless schedules. Thus, all strict schedules are cascadeless, and all cascadeless schedules are recoverable.

21.5 Characterizing Schedules Based on Serializability

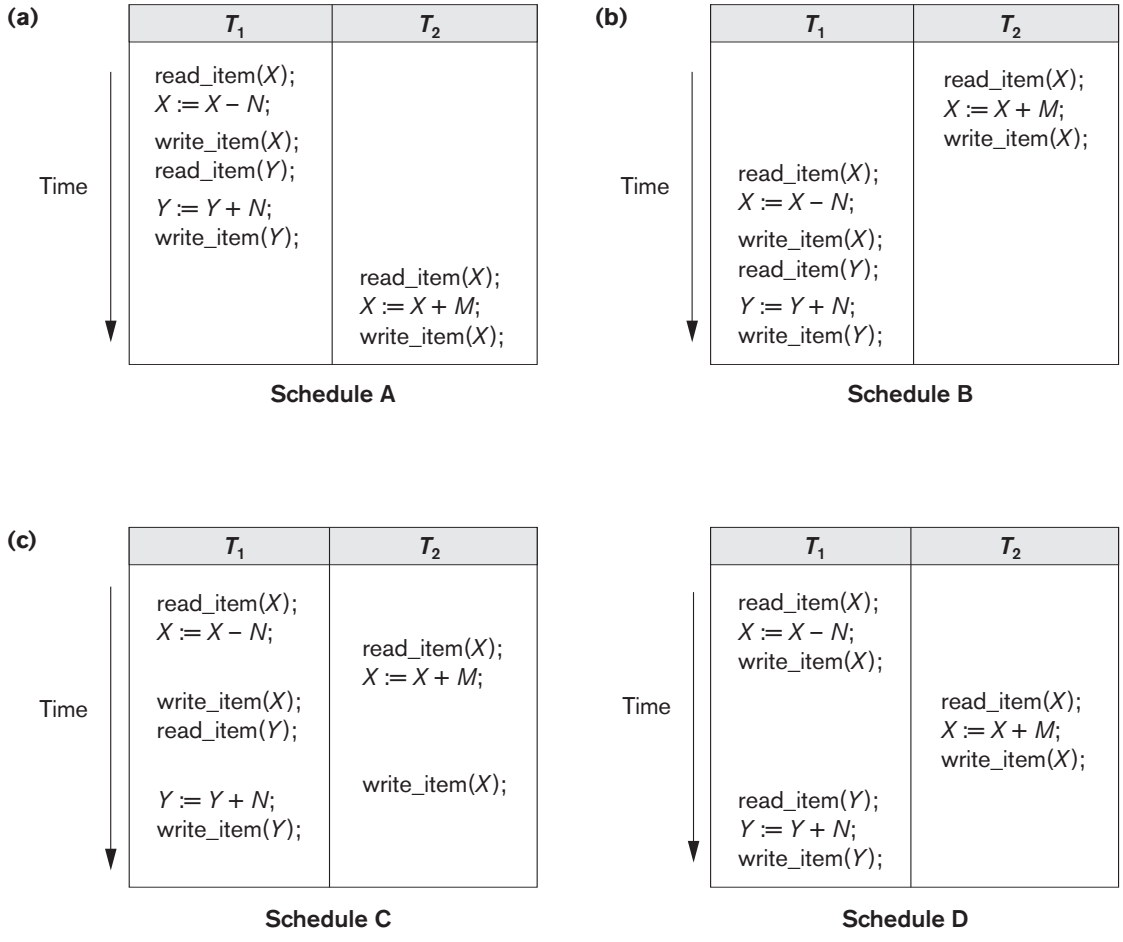
In the previous section, we characterized schedules based on their recoverability properties. Now we characterize the types of schedules that are always considered to be *correct* when concurrent transactions are executing. Such schedules are known as *serializable schedules*. Suppose that two users—for example, two airline reservations agents—submit to the DBMS transactions T_1 and T_2 in Figure 21.2 at approximately the same time. If no interleaving of operations is permitted, there are only two possible outcomes:

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 two schedules—called *serial schedules*—are shown in Figure 21.5(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 21.5(c). The concept of **serializability of schedules** is used to identify which schedules are correct when transaction executions have interleaving of their operations in the schedules. This section defines serializability and discusses how it may be used in practice.

Figure 21.5
Examples of serial and nonserial schedules involving transactions T_1 and T_2. (a) Serial schedule A: T_1 followed by T_2. (b) Serial schedule B: T_2 followed by T_1. (c) Two nonserial schedules C and D with interleaving of operations.



21.5.1 Serial, Nonserial, and Conflict-Serializable Schedules

Schedules A and B in Figure 21.5(a) and (b) are called *serial* 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 21.5(a), and T_2 and then T_1 in Figure 21.5(b). Schedules C and D in Figure 21.5(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**. Therefore, in a serial schedule, only one transaction at a time is active—the commit (or abort) of the active transaction initiates execution of the next transaction. No interleaving occurs in a serial schedule. One reasonable assumption we can make, if we consider the transactions to be *independent*, is that *every serial schedule is considered correct*. We can assume this because every transaction is assumed to be correct if executed on its own (according to the *consistency preservation* property of Section 21.3). Hence, it does not matter which transaction is executed first. As long as every transaction is executed from beginning to end in isolation 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 by prohibiting 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. Additionally, if some transaction T is quite long, the other transactions must wait for T to complete all its operations before starting. Hence, serial schedules are *considered unacceptable* in practice. However, if we can determine which other schedules are *equivalent* to a serial schedule, we can allow these schedules to occur.

To illustrate our discussion, consider the schedules in Figure 21.5, and assume that the initial values of database items are $X = 90$ and $Y = 90$ and that $N = 3$ and $M = 2$. After executing transactions T_1 and T_2, 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 (which is the same as Figure 21.3(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 21.1.3; 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 nonserial 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.

The definition of *serializable schedule* is as follows: 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 schedules* 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 schedule equivalence. The simplest but least satisfactory definition 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 21.6, schedules S_1 and S_2 will produce the same final database state if they execute on a database with an initial value of $X = 100$; however, for other initial values of X, the schedules are *not* result equivalent. Additionally, these schedules execute different transactions, so they definitely should not be considered equivalent. Hence, result equivalence alone cannot be used to define equivalence of schedules. The safest and most general approach to defining schedule equivalence is not to make any assumptions 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 used: *conflict equivalence* and *view equivalence*. We discuss conflict equivalence next, which is the more commonly used definition.

The definition of *conflict equivalence* of schedules is as follows: 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 21.4.1 that two operations in a schedule are said to *conflict* if they belong to different transactions, access the same database item, and either both are write_item operations or one is a write_item and the other a read_item. If two conflicting operations are applied in *different orders* in two schedules, the effect can be different on the database or on the transactions in the schedule, and hence the schedules are not conflict equivalent. For example, as we discussed in Section 21.4.1, if a read and write operation occur in the order $r_1(X)$, $w_2(X)$ in schedule S_1, and in the reverse order $w_2(X)$, $r_1(X)$ in schedule S_2, the value read by $r_1(X)$ can be different in the two schedules. Similarly, if two write operations

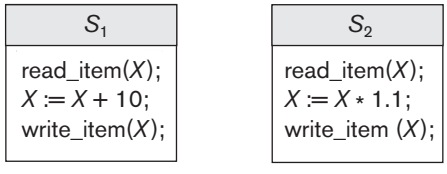


Figure 21.6
Two schedules that are result equivalent for the initial value of $X = 100$ but are not result equivalent in general.

occur in the order $w_1(X)$, $w_2(X)$ in S_1, and in the reverse order $w_2(X)$, $w_1(X)$ in S_2, the next $r(X)$ operation in the two schedules will read potentially different values; or if these are the last operations writing item X in the schedules, the final value of item X in the database will be different.

Using the notion of conflict equivalence, we define a schedule S to be **conflict serializable**[12] 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 in Figure 21.5(c) is equivalent to the serial schedule A in Figure 21.5(a). In both schedules, the 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. Additionally, T_1 is the last transaction to write 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. Therefore, 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 in Figure 21.5(c) is not equivalent to either of the two possible serial schedules A and B, and hence is *not serializable*. Trying to reorder the operations of schedule C to find an equivalent serial schedule fails because $r_2(X)$ and $w_1(X)$ conflict, which means that we cannot move $r_2(X)$ 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 21.5.4.

21.5.2 Testing for Conflict Serializability of a Schedule

There is a simple algorithm for determining whether a particular schedule is conflict serializable or not. Most concurrency control methods do *not* actually test for serializability. Rather protocols, or rules, are developed that guarantee that any schedule that follows these rules will be serializable. We discuss the algorithm for testing conflict serializability of schedules here to gain a better understanding of these concurrency control protocols, which are discussed in Chapter 22.

Algorithm 21.1 can be used to test a schedule for conflict serializability. The algorithm looks at only the read_item and write_item operations in a schedule to construct a **precedence graph** (or **serialization graph**), which is a **directed graph** $G = (N, E)$ that consists of a set of nodes $N = \{T_1, T_2, ..., T_n\}$ and a set of directed edges $E = \{e_1, e_2, ..., e_m\}$. 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 $(T_j \rightarrow T_k)$, $1 \le j \le n$, $1 \le k \le n$, where T_j is the **starting node** of e_i and T_k is the **ending node** of e_i. Such an edge from node T_j to

[12]We will use *serializable* to mean conflict serializable. Another definition of serializable used in practice (see Section 21.6) is to have repeatable reads, no dirty reads, and no phantom records (see Section 22.7.1 for a discussion on phantoms).

node T_k is created by the algorithm if one of the operations in T_j appears in the schedule before some *conflicting operation* in T_k.

Algorithm 21.1. 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 T_i executes a write_item(X), create an edge $(T_i \rightarrow T_j)$ in the precedence graph.
3. For each case in S where T_j executes a 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) after T_i executes a write_item(X), 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 described in Algorithm 21.1. 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** $C = ((T_j \rightarrow T_k), (T_k \rightarrow T_p), \ldots, (T_i \rightarrow T_j))$ 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 (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'.[13] 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. Figure 21.7 shows such labels on the edges.

In general, several serial schedules can be equivalent to S if the precedence graph for S has no cycle. However, if the precedence graph has a cycle, it is easy to show that we cannot create any equivalent serial schedule, so S is not serializable. The precedence graphs created for schedules A to D, respectively, in Figure 21.5 appear in Figure 21.7(a) to (d). The 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 21.8. Figure 21.8(a) shows the read_item and write_item operations in each transaction. Two schedules E and F for these transactions are shown in Figure 21.8(b) and (c),

[13] This process of ordering the nodes of an acrylic graph is known as *topological sorting*.

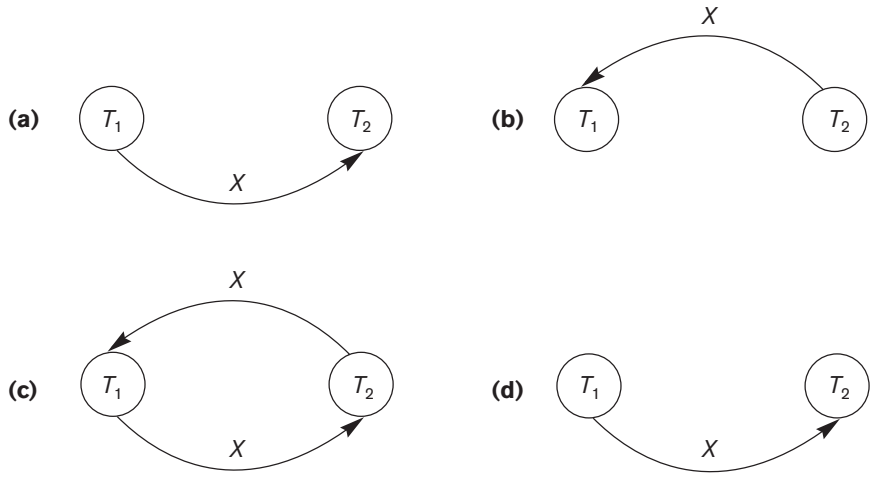


Figure 21.7
Constructing the precedence graphs for schedules A to D from Figure 21.5 to test for conflict serializability. (a) Precedence graph for serial schedule A. (b) Precedence graph for serial schedule B. (c) Precedence graph for schedule C (not serializable). (d) Precedence graph for schedule D (serializable, equivalent to schedule A).

respectively, and the precedence graphs for schedules E and F are shown in parts (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 21.8(e). Although only one equivalent serial schedule exists for *F*, in general there may be more than one equivalent serial schedule for a serializable schedule. Figure 21.8(f) shows a precedence graph representing a schedule that has two equivalent serial schedules. To find an equivalent serial schedule, start with a node that does not have any incoming edges, and then make sure that the node order for every edge is not violated.

21.5.3 How Serializability Is Used for Concurrency Control

As we discussed earlier, saying that a schedule *S* is (conflict) serializable—that is, *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, or for another transaction to terminate, thus slowing down processing considerably. A serializable schedule gives 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—which are usually executed as processes by the operating system—is typically determined by the operating system scheduler, which allocates resources to

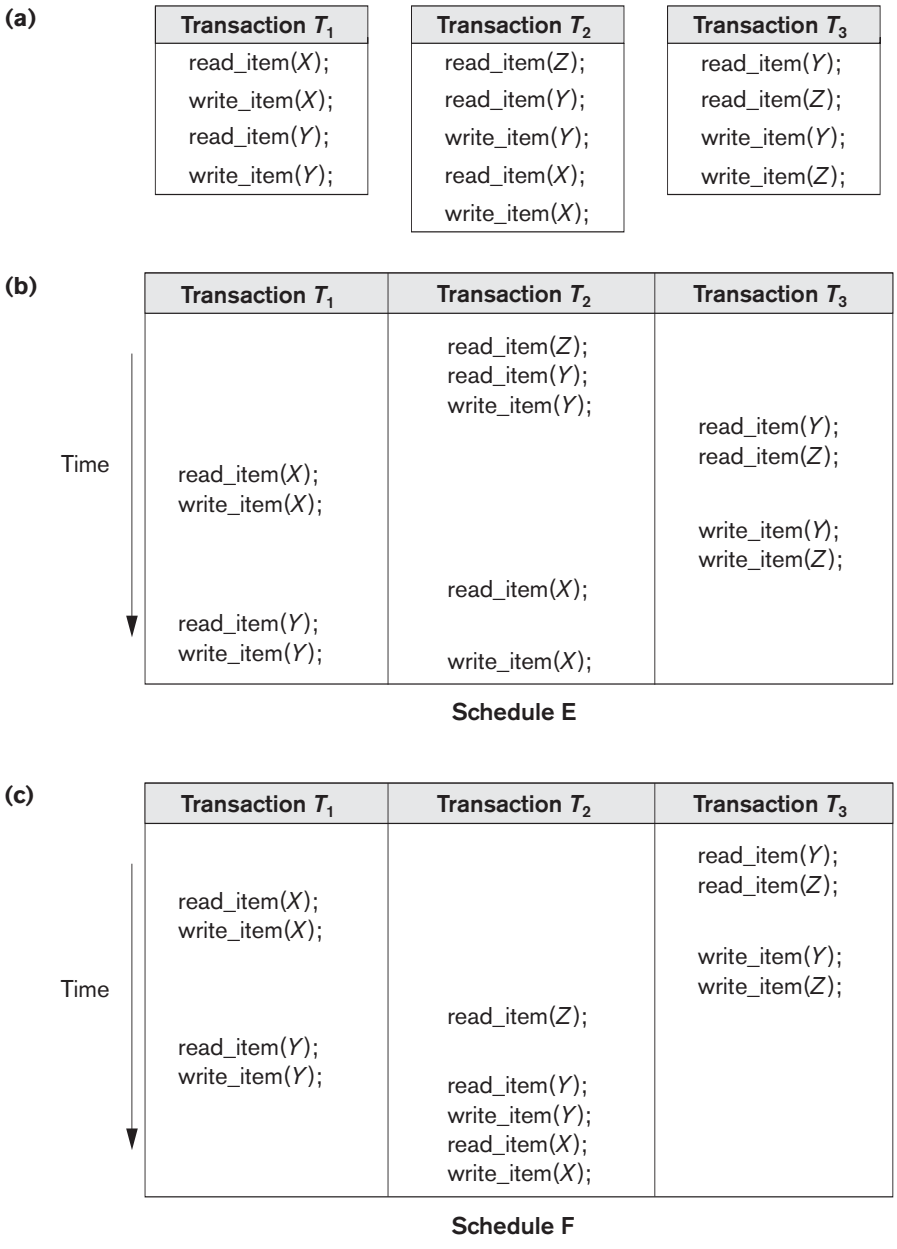
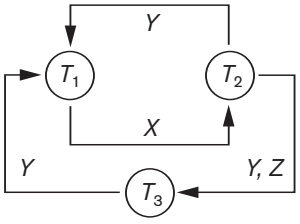


Figure 21.8
Another example of serializability testing. (a) The read and write operations of three transactions T_1, T_2, and T_3. (b) Schedule E. (c) Schedule F.

all processes. Factors such as system load, time of transaction submission, and priorities of processes contribute to the ordering of operations in a schedule. Hence, it is difficult to determine how the operations of a schedule will be interleaved beforehand to ensure serializability.

(d)



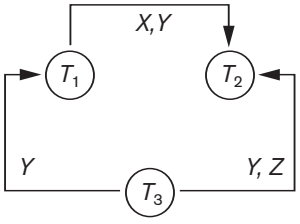
Equivalent serial schedules

None

Reason

Cycle $X(T_1 \rightarrow T_2), Y(T_2 \rightarrow T_1)$
Cycle $X(T_1 \rightarrow T_2), YZ\ (T_2 \rightarrow T_3), Y(T_3 \rightarrow T_1)$

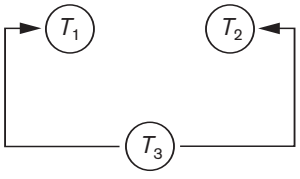
(e)



Equivalent serial schedules

$T_3 \rightarrow T_1 \rightarrow T_2$

(f)



Equivalent serial schedules

$T_3 \rightarrow T_1 \rightarrow T_2$

$T_3 \rightarrow T_2 \rightarrow T_1$

Figure 21.8 (continued)
Another example of serializability testing.
(d) Precedence graph for schedule E.
(e) Precedence graph for schedule F.
(f) 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 or protocols that ensure serializability, without having to test the schedules themselves. The approach taken in most commercial DBMSs is to design **protocols** (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*.

Another problem appears here: 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 21.4.1 that the *committed projection* $C(S)$ of a schedule *S* includes only the operations in *S* that belong to committed transactions. We can theoretically 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 22, 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, and enforcing an additional condition that guarantees serializability. This is used in the majority of commercial DBMSs. Other protocols have been proposed;[14] these include *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; *multiversion protocols*, which are based on maintaining multiple versions of data items; and *optimistic* (also called *certification* or *validation*) *protocols*, which check for possible serializability violations after the transactions terminate but before they are permitted to commit.

21.5.4 View Equivalence and View Serializability

In Section 21.5.1 we defined the concepts of conflict equivalence of schedules and 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 S and S' 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** (or **no blind writes**) holds on all transactions in the schedule. 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. A **blind write** is a write operation in a transaction T on an item X that is not dependent on the value of X, so it is not preceded by a read of X in the transaction T.

[14]These other protocols have not been incorporated much into commercial systems; most relational DBMSs use some variation of the two-phase locking protocol.

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 its old value from the database. This is possible when *blind writes* are allowed, and it is illustrated by the following schedule S_g of three transactions T_1: $r_1(X)$; $w_1(X)$; T_2: $w_2(X)$; and T_3: $w_3(X)$:

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

In S_g the operations $w_2(X)$ and $w_3(X)$ are blind writes, since T_2 and T_3 do not read the value of X. The schedule S_g is view serializable, since it is view equivalent to the serial schedule T_1, T_2, T_3. However, S_g 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-hard, meaning that finding an efficient polynomial time algorithm for this problem is highly unlikely.

21.5.5 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 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: $r_2(Y)$; $Y := Y - 20$; $w_2(Y)$; $r_2(X)$; $X := X + 20$; $w_2(X)$;

Consider the following nonserializable schedule S_h for the two transactions:

S_h: $r_1(X)$; $w_1(X)$; $r_2(Y)$; $w_2(Y)$; $r_1(Y)$; $w_1(Y)$; $r_2(X)$; $w_2(X)$;

With the additional knowledge, or **semantics**, that the operations between each $r_i(I)$ and $w_i(I)$ 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 by a particular transaction T_i on a particular item I is not interrupted by conflicting operations. Hence, the schedule S_h is considered to be correct even though it is not serializable. 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. Also, in certain domains of applications such as computer aided design (CAD) of complex systems like aircraft,

design transactions last over a long time period. In such applications, more relaxed schemes of concurrency control have been proposed to maintain consistency of the database.

21.6 Transaction Support in SQL

In this section, we give a brief introduction to transaction support in SQL. There are many more details, and the newer standards have more commands for transaction processing. The basic definition of an SQL transaction is similar to our already defined concept of a transaction. That is, it is a logical unit of work and is guaranteed to be atomic. A single SQL statement is always considered to be atomic—either it completes execution without an error or it fails and leaves the database unchanged.

With SQL, there is no explicit Begin_Transaction statement. Transaction initiation is done implicitly when particular SQL statements are encountered. However, every transaction must have an explicit end statement, which is either a COMMIT or a ROLLBACK. Every transaction has certain characteristics attributed to it. These characteristics are specified by a SET TRANSACTION statement in SQL. The characteristics are the *access mode*, the *diagnostic area size*, and the *isolation level*.

The **access mode** can be specified as READ ONLY or READ WRITE. The default is READ WRITE, unless the isolation level of READ UNCOMMITTED is specified (see below), in which case READ ONLY is assumed. A mode of READ WRITE allows select, update, insert, delete, and create commands to be executed. A mode of READ ONLY, as the name implies, is simply for data retrieval.

The **diagnostic area size** option, DIAGNOSTIC SIZE n, specifies an integer value n, which indicates the number of conditions that can be held simultaneously in the diagnostic area. These conditions supply feedback information (errors or exceptions) to the user or program on the n most recently executed SQL statement.

The **isolation level** option is specified using the statement ISOLATION LEVEL <isolation>, where the value for <isolation> can be READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, or SERIALIZABLE.[15] The default isolation level is SERIALIZABLE, although some systems use READ COMMITTED as their default. The use of the term SERIALIZABLE here is based on not allowing violations that cause dirty read, unrepeatable read, and phantoms,[16] and it is thus not identical to the way serializability was defined earlier in Section 21.5. If a transaction executes at a lower isolation level than SERIALIZABLE, then one or more of the following three violations may occur:

1. **Dirty read.** A transaction T_1 may read the update of a transaction T_2, which has not yet committed. If T_2 fails and is aborted, then T_1 would have read a value that does not exist and is incorrect.

[15]These are similar to the *isolation levels* discussed briefly at the end of Section 21.3.

[16]The dirty read and unrepeatable read problems were discussed in Section 21.1.3. Phantoms are discussed in Section 22.7.1.

2. **Nonrepeatable read.** A transaction T_1 may read a given value from a table. If another transaction T_2 later updates that value and T_1 reads that value again, T_1 will see a different value.
3. **Phantoms.** A transaction T_1 may read a set of rows from a table, perhaps based on some condition specified in the SQL WHERE-clause. Now suppose that a transaction T_2 inserts a new row that also satisfies the WHERE-clause condition used in T_1, into the table used by T_1. If T_1 is repeated, then T_1 will see a phantom, a row that previously did not exist.

Table 21.1 summarizes the possible violations for the different isolation levels. An entry of *Yes* indicates that a violation is possible and an entry of *No* indicates that it is not possible. READ UNCOMMITTED is the most forgiving, and SERIALIZABLE is the most restrictive in that it avoids all three of the problems mentioned above.

A sample SQL transaction might look like the following:

```
EXEC SQL WHENEVER SQLERROR GOTO UNDO;
EXEC SQL SET TRANSACTION
    READ WRITE
    DIAGNOSTIC SIZE 5
    ISOLATION LEVEL SERIALIZABLE;
EXEC SQL INSERT INTO EMPLOYEE (Fname, Lname, Ssn, Dno, Salary)
    VALUES ('Robert', 'Smith', '991004321', 2, 35000);
EXEC SQL UPDATE EMPLOYEE
    SET Salary = Salary * 1.1 WHERE Dno = 2;
EXEC SQL COMMIT;
GOTO THE_END;
UNDO: EXEC SQL ROLLBACK;
THE_END: ... ;
```

The above transaction consists of first inserting a new row in the EMPLOYEE table and then updating the salary of all employees who work in department 2. If an error occurs on any of the SQL statements, the entire transaction is rolled back. This implies that any updated salary (by this transaction) would be restored to its previous value and that the newly inserted row would be removed.

As we have seen, SQL provides a number of transaction-oriented features. The DBA or database programmers can take advantage of these options to try improving

Table 21.1 Possible Violations Based on Isolation Levels as Defined in SQL

	Type of Violation		
Isolation Level	**Dirty Read**	**Nonrepeatable Read**	**Phantom**
READ UNCOMMITTED	Yes	Yes	Yes
READ COMMITTED	No	Yes	Yes
REPEATABLE READ	No	No	Yes
SERIALIZABLE	No	No	No

transaction performance by relaxing serializability if that is acceptable for their applications.

21.7 Summary

In this chapter we discussed DBMS concepts for transaction processing. 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.

Next we introduced the typical states that a transaction passes through during execution, and discussed several concepts that are used in recovery and concurrency control methods. The system log keeps track of database accesses, and the system uses this information to recover from failures. A transaction either succeeds and reaches its commit point or it fails and has to be rolled back. A committed transaction has its changes permanently recorded in the database. We presented an overview of the desirable properties of transactions—atomicity, consistency preservation, isolation, and durability—which are often referred to as the ACID properties.

Then we defined a schedule (or history) as an execution sequence of the operations of several transactions with possible interleaving. We characterized schedules in terms of their recoverability. Recoverable schedules ensure that, once a transaction commits, it never needs to be undone. Cascadeless schedules add an additional condition to ensure that no aborted transaction requires the cascading abort of other transactions. Strict schedules provide an even stronger condition that allows a simple recovery scheme consisting of restoring the old values of items that have been changed by an aborted transaction.

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, which led to definitions for conflict serializability and view serializability. A serializable schedule is considered correct. We presented an algorithm for testing the (conflict) serializability of a schedule. 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. Finally, we gave a brief overview of how transaction concepts are used in practice within SQL.

Review Questions

21.1. What is meant by the concurrent execution of database transactions in a multiuser system? Discuss why concurrency control is needed, and give informal examples.

21.2. Discuss the different types of failures. What is meant by catastrophic failure?

21.3. Discuss the actions taken by the read_item and write_item operations on a database.

21.4. Draw a state diagram and discuss the typical states that a transaction goes through during execution.

21.5. What is the system log used for? What are the typical kinds of records in a system log? What are transaction commit points, and why are they important?

21.6. Discuss the atomicity, durability, isolation, and consistency preservation properties of a database transaction.

21.7. What is a schedule (history)? Define the concepts of recoverable, cascadeless, and strict schedules, and compare them in terms of their recoverability.

21.8. Discuss the different measures of transaction equivalence. What is the difference between conflict equivalence and view equivalence?

21.9. What is a serial schedule? What is a serializable schedule? Why is a serial schedule considered correct? Why is a serializable schedule considered correct?

21.10. What is the difference between the constrained write and the unconstrained write assumptions? Which is more realistic?

21.11. 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?

21.12. Describe the four levels of isolation in SQL.

21.13. Define the violations caused by each of the following: dirty read, nonrepeatable read, and phantoms.

Exercises

21.14. Change transaction T_2 in Figure 21.2(b) 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 21.3(a) and (b), 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)?

21.15. Repeat Exercise 21.14, adding a check in T_1 so that Y does not exceed 90.

21.16. Add the operation commit at the end of each of the transactions T_1 and T_2 in Figure 21.2, and then list all possible schedules for the modified transactions. Determine which of the schedules are recoverable, which are cascadeless, and which are strict.

21.17. List all possible schedules for transactions T_1 and T_2 in Figure 21.2, and determine which are conflict serializable (correct) and which are not.

21.18. How many *serial* schedules exist for the three transactions in Figure 21.8(a)? What are they? What is the total number of possible schedules?

21.19. Write a program to create all possible schedules for the three transactions in Figure 21.8(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.

21.20. Why is an explicit transaction end statement needed in SQL but not an explicit begin statement?

21.21. Describe situations where each of the different isolation levels would be useful for transaction processing.

21.22. Which of the following schedules is (conflict) serializable? For each serializable schedule, determine the equivalent serial schedules.

a. $r_1(X); r_3(X); w_1(X); r_2(X); w_3(X);$

b. $r_1(X); r_3(X); w_3(X); w_1(X); r_2(X);$

c. $r_3(X); r_2(X); w_3(X); r_1(X); w_1(X);$

d. $r_3(X); r_2(X); r_1(X); w_3(X); w_1(X);$

21.23. Consider the three transactions T_1, T_2, and T_3, and the schedules S_1 and S_2 given below. Draw the serializability (precedence) graphs for S_1 and S_2, and state whether each schedule is serializable or not. If a schedule is serializable, write down the equivalent serial schedule(s).

T_1: $r_1(X); r_1(Z); w_1(X);$
T_2: $r_2(Z); r_2(Y); w_2(Z); w_2(Y);$
T_3: $r_3(X); r_3(Y); w_3(Y);$
S_1: $r_1(X); r_2(Z); r_1(Z); r_3(X); r_3(Y); w_1(X); w_3(Y); r_2(Y); w_2(Z); w_2(Y);$
S_2: $r_1(X); r_2(Z); r_3(X); r_1(Z); r_2(Y); r_3(Y); w_1(X); w_2(Z); w_3(Y); w_2(Y);$

21.24. Consider schedules S_3, S_4, and S_5 below. Determine whether each schedule is strict, cascadeless, recoverable, or nonrecoverable. (Determine the strictest recoverability condition that each schedule satisfies.)

S_3: $r_1(X); r_2(Z); r_1(Z); r_3(X); r_3(Y); w_1(X); c_1; w_3(Y); c_3; r_2(Y); w_2(Z); w_2(Y); c_2;$
S_4: $r_1(X); r_2(Z); r_1(Z); r_3(X); r_3(Y); w_1(X); w_3(Y); r_2(Y); w_2(Z); w_2(Y); c_1; c_2; c_3;$
S_5: $r_1(X); r_2(Z); r_3(X); r_1(Z); r_2(Y); r_3(Y); w_1(X); c_1; w_2(Z); w_3(Y); w_2(Y); c_3; c_2;$

Selected Bibliography

The concept of serializability and related ideas to maintain consistency in a database were introduced in Gray et al. (1975). The concept of the database transaction was first discussed in Gray (1981). Gray won the coveted ACM Turing Award in 1998 for his work on database transactions and implementation of transactions in relational DBMSs. Bernstein, Hadzilacos, and Goodman (1988) focus on concurrency control and recovery techniques in both centralized and distributed database systems; it is an excellent reference. Papadimitriou (1986) offers a more theoretical perspective. A large reference book of more than a thousand pages by Gray and Reuter (1993) offers a more practical perspective of transaction processing concepts and techniques. Elmagarmid (1992) offers collections of research papers on transaction processing for advanced applications. Transaction support in SQL is described in Date and Darwen (1997). View serializability is defined in Yannakakis (1984). Recoverability of schedules and reliability in databases is discussed in Hadzilacos (1983, 1988).

chapter 22

Concurrency Control Techniques

In this chapter we discuss a number of concurrency control techniques that are used to ensure the noninterference or isolation property of concurrently executing transactions. Most of these techniques ensure serializability of schedules—which we defined in Section 21.5—using **concurrency control protocols** (sets of rules) that guarantee serializability. One important set of protocols—known as *two-phase locking protocols*—employ the technique of **locking** data items to prevent multiple transactions from accessing the items concurrently; a number of locking protocols are described in Sections 22.1 and 22.3.2. Locking protocols are used in most commercial DBMSs. Another set of concurrency control protocols use **timestamps**. A timestamp is a unique identifier for each transaction, generated by the system. Timestamps values are generated in the same order as the transaction start times. Concurrency control protocols that use timestamp ordering to ensure serializability are introduced in Section 22.2. In Section 22.3 we discuss **multiversion** concurrency control protocols that use multiple versions of a data item. One multiversion protocol extends timestamp order to multiversion timestamp ordering (Section 22.3.1), and another extends two-phase locking (Section 22.3.2). In Section 22.4 we present a protocol based on the concept of **validation** or **certification** of a transaction after it executes its operations; these are sometimes called **optimistic protocols**, and also assume that multiple versions of a data item can exist.

Another factor that affects concurrency control is the **granularity** of the data items—that is, what portion of the database a data item represents. An item can be as small as a single attribute (field) value or as large as a disk block, or even a whole file or the entire database. We discuss granularity of items and a multiple granularity concurrency control protocol, which is an extension of two-phase locking, in Section 22.5. In Section 22.6 we describe concurrency control issues that arise when

indexes are used to process transactions, and in Section 22.7 we discuss some additional concurrency control concepts. Section 22.8 summarizes the chapter.

It is sufficient to read Sections 22.1, 22.5, 22.6, and 22.7, and possibly 22.3.2, if your main interest is an introduction to the concurrency control techniques that are based on locking, which are used most often in practice. The other techniques are mainly of theoretical interest.

22.1 Two-Phase Locking Techniques for Concurrency Control

Some of the main techniques used to control concurrent execution of transactions are based on the concept of locking data items. A **lock** is a variable associated with a data item that describes the status of the item with respect to possible operations that can be applied to it. Generally, there is one lock for each data item in the database. Locks are used as a means of synchronizing the access by concurrent transactions to the database items. In Section 22.1.1 we discuss the nature and types of locks. Then, in Section 22.1.2 we present protocols that use locking to guarantee serializability of transaction schedules. Finally, in Section 22.1.3 we describe two problems associated with the use of locks—deadlock and starvation—and show how these problems are handled in concurrency control protocols.

22.1.1 Types of Locks and System Lock Tables

Several types of locks are used in concurrency control. To introduce locking concepts gradually, first we discuss binary locks, which are simple, but are also *too restrictive for database concurrency control purposes*, and so are not used in practice. Then we discuss *shared/exclusive* locks—also known as *read/write* locks—which provide more general locking capabilities and are used in practical database locking schemes. In Section 22.3.2 we describe an additional type of lock called a *certify lock*, and show how it can be used to improve performance of locking protocols.

Binary Locks. A **binary lock** can have two **states** or **values:** locked and unlocked (or 1 and 0, for simplicity). A distinct lock is associated with each database item X. If the value of the lock on X is 1, item X *cannot be accessed* by a database operation that requests the item. If the value of the lock on X is 0, the item can be accessed when requested, and the lock value is changed to 1. We refer to the current value (or state) of the lock associated with item X as **lock**(X).

Two operations, lock_item and unlock_item, are used with binary locking. A transaction requests access to an item X by first issuing a **lock_item**(X) operation. If LOCK(X) = 1, the transaction is forced to wait. If LOCK(X) = 0, it is set to 1 (the transaction **locks** the item) and the transaction is allowed to access item X. When the transaction is through using the item, it issues an **unlock_item**(X) operation, which sets LOCK(X) back to 0 (**unlocks** the item) so that X may be accessed by other transactions. Hence, a binary lock enforces **mutual exclusion** on the data item. A description of the lock_item(X) and unlock_item(X) operations is shown in Figure 22.1.

```
lock_item(X):
B:  if LOCK(X) = 0              (* item is unlocked *)
        then LOCK(X) ←1         (* lock the item *)
    else
        begin
        wait (until LOCK(X) = 0
            and the lock manager wakes up the transaction);
        go to B
        end;
unlock_item(X):
    LOCK(X) ← 0;                (* unlock the item *)
    if any transactions are waiting
        then wakeup one of the waiting transactions;
```

Figure 22.1
Lock and unlock operations for binary locks.

Notice that the lock_item and unlock_item operations must be implemented as indivisible units (known as **critical sections** in operating systems); that is, no interleaving should be allowed once a lock or unlock operation is started until the operation terminates or the transaction waits. In Figure 22.1, the wait command within the lock_item(X) operation is usually implemented by putting the transaction in a waiting queue for item X until X is unlocked and the transaction can be granted access to it. Other transactions that also want to access X are placed in the same queue. Hence, the wait command is considered to be outside the lock_item operation.

It is quite simple to implement a binary lock; all that is needed is a binary-valued variable, LOCK, associated with each data item X in the database. In its simplest form, each lock can be a record with three fields: <Data_item_name, LOCK, Locking_transaction> plus a queue for transactions that are waiting to access the item. The system needs to maintain *only these records for the items that are currently locked* in a **lock table**, which could be organized as a hash file on the item name. Items not in the lock table are considered to be unlocked. The DBMS has a **lock manager subsystem** to keep track of and control access to locks.

If the simple binary locking scheme described here is used, every transaction must obey the following rules:

1. A transaction T must issue the operation lock_item(X) before any read_item(X) or write_item(X) operations are performed in T.
2. A transaction T must issue the operation unlock_item(X) after all read_item(X) and write_item(X) operations are completed in T.
3. A transaction T will not issue a lock_item(X) operation if it already holds the lock on item X.[1]
4. A transaction T will not issue an unlock_item(X) operation unless it already holds the lock on item X.

[1]This rule may be removed if we modify the lock_item (X) operation in Figure 22.1 so that if the item is currently locked *by the requesting transaction*, the lock is granted.

These rules can be enforced by the lock manager module of the DBMS. Between the lock_item(X) and unlock_item(X) operations in transaction T, T is said to **hold the lock** on item X. At most one transaction can hold the lock on a particular item. Thus no two transactions can access the same item concurrently.

Shared/Exclusive (or Read/Write) Locks. The preceding binary locking scheme is too restrictive for database items because at most, one transaction can hold a lock on a given item. We should allow several transactions to access the same item X if they all access X for *reading purposes only*. This is because read operations on the same item by different transactions are not conflicting (see Section 21.4.1). However, if a transaction is to write an item X, it must have exclusive access to X. For this purpose, a different type of lock called a **multiple-mode lock** is used. In this scheme—called **shared/exclusive** or **read/write** locks—there are three locking operations: read_lock(X), write_lock(X), and unlock(X). A lock associated with an item X, LOCK(X), now has three possible states: *read-locked*, *write-locked*, or *unlocked*. A **read-locked item** is also called **share-locked** because other transactions are allowed to read the item, whereas a **write-locked item** is called **exclusive-locked** because a single transaction exclusively holds the lock on the item.

One method for implementing the preceding operations on a read/write lock is to keep track of the number of transactions that hold a shared (read) lock on an item in the lock table. Each record in the lock table will have four fields: <Data_item_name, LOCK, No_of_reads, Locking_transaction(s)>. Again, to save space, the system needs to maintain lock records only for locked items in the lock table. The value (state) of LOCK is either read-locked or write-locked, suitably coded (if we assume no records are kept in the lock table for unlocked items). If LOCK(X)=write-locked, the value of locking_transaction(s) is a single transaction that holds the exclusive (write) lock on X. If LOCK(X)=read-locked, the value of locking transaction(s) is a list of one or more transactions that hold the shared (read) lock on X. The three operations read_lock(X), write_lock(X), and unlock(X) are described in Figure 22.2.[2] As before, each of the three locking operations should be considered indivisible; no interleaving should be allowed once one of the operations is started until either the operation terminates by granting the lock or the transaction is placed in a waiting queue for the item.

When we use the shared/exclusive locking scheme, the system must enforce the following rules:

1. A transaction T must issue the operation read_lock(X) or write_lock(X) before any read_item(X) operation is performed in T.
2. A transaction T must issue the operation write_lock(X) before any write_item(X) operation is performed in T.

[2]These algorithms do not allow *upgrading* or *downgrading* of locks, as described later in this section. The reader can extend the algorithms to allow these additional operations.

```
read_lock(X):
B:  if LOCK(X) = "unlocked"
        then begin LOCK(X) ← "read-locked";
                no_of_reads(X) ← 1
                end
    else if LOCK(X) = "read-locked"
        then no_of_reads(X) ← no_of_reads(X) + 1
    else begin
            wait (until LOCK(X) = "unlocked"
                and the lock manager wakes up the transaction);
            go to B
            end;
write_lock(X):
B:  if LOCK(X) = "unlocked"
        then LOCK(X) ← "write-locked"
    else begin
            wait (until LOCK(X) = "unlocked"
                and the lock manager wakes up the transaction);
            go to B
            end;
unlock (X):
    if LOCK(X) = "write-locked"
        then begin LOCK(X) ← "unlocked";
                wakeup one of the waiting transactions, if any
                end
    else it LOCK(X) = "read-locked"
        then begin
                no_of_reads(X) ← no_of_reads(X) –1;
                if no_of_reads(X) = 0
                    then begin LOCK(X) = "unlocked";
                            wakeup one of the waiting transactions, if any
                            end
                end;
```

Figure 22.2
Locking and unlocking operations for two-mode (read-write or shared-exclusive) locks.

3. A transaction T must issue the operation unlock(X) after all read_item(X) and write_item(X) operations are completed in T.[3]
4. A transaction T will not issue a read_lock(X) operation if it already holds a read (shared) lock or a write (exclusive) lock on item X. This rule may be relaxed, as we discuss shortly.

[3]This rule may be relaxed to allow a transaction to unlock an item, then lock it again later.

5. A transaction T will not issue a write_lock(X) operation if it already holds a read (shared) lock or write (exclusive) lock on item X. This rule may also be relaxed, as we discuss shortly.
6. A transaction T will not issue an unlock(X) operation unless it already holds a read (shared) lock or a write (exclusive) lock on item X.

Conversion of Locks. Sometimes it is desirable to relax conditions 4 and 5 in the preceding list in order to allow **lock conversion**; that is, a transaction that already holds a lock on item X is allowed under certain conditions to **convert** the lock from one locked state to another. For example, it is possible for a transaction T to issue a read_lock(X) and then later to **upgrade** the lock by issuing a write_lock(X) operation. If T is the only transaction holding a read lock on X at the time it issues the write_lock(X) operation, the lock can be upgraded; otherwise, the transaction must wait. It is also possible for a transaction T to issue a write_lock(X) and then later to **downgrade** the lock by issuing a read_lock(X) operation. When upgrading and downgrading of locks is used, the lock table must include transaction identifiers in the record structure for each lock (in the locking_transaction(s) field) to store the information on which transactions hold locks on the item. The descriptions of the read_lock(X) and write_lock(X) operations in Figure 22.2 must be changed appropriately to allow for lock upgrading and downgrading. We leave this as an exercise for the reader.

Using binary locks or read/write locks in transactions, as described earlier, does not guarantee serializability of schedules on its own. Figure 22.3 shows an example where the preceding locking rules are followed but a nonserializable schedule may result. This is because in Figure 22.3(a) the items Y in T_1 and X in T_2 were unlocked too early. This allows a schedule such as the one shown in Figure 22.3(c) to occur, which is not a serializable schedule and hence gives incorrect results. To guarantee serializability, we must follow *an additional protocol* concerning the positioning of locking and unlocking operations in every transaction. The best-known protocol, two-phase locking, is described in the next section.

22.1.2 Guaranteeing Serializability by Two-Phase Locking

A transaction is said to follow the **two-phase locking protocol** if *all* locking operations (read_lock, write_lock) precede the *first* unlock operation in the transaction.[4] Such a transaction can be divided into two phases: an **expanding** or **growing (first) phase**, during which new locks on items can be acquired but none can be released; and a **shrinking (second) phase**, during which existing locks can be released but no new locks can be acquired. If lock conversion is allowed, then upgrading of locks (from read-locked to write-locked) must be done during the expanding phase, and downgrading of locks (from write-locked to read-locked) must be done in the

[4]This is unrelated to the two-phase commit protocol for recovery in distributed databases (see Chapter 25).

(a)

T_1	T_2

read_lock(Y);
read_item(Y);
unlock(Y);
write_lock(X);
read_item(X);
$X := X + Y$;
write_item(X);
unlock(X);
	read_lock(X);
	read_item(X);
	unlock(X);
	write_lock(Y);
	read_item(Y);
	$Y := X + Y$;
	write_item(Y);
	unlock(Y);

(b)

Initial values: X=20, Y=30

Result serial schedule T_1 followed by T_2: X=50, Y=80

Result of serial schedule T_2 followed by T_1: X=70, Y=50

(c)

T_1	T_2

read_lock(Y);
read_item(Y);
unlock(Y);	

	read_lock(X);
	read_item(X);
	unlock(X);
	write_lock(Y);
	read_item(Y);
	$Y := X + Y$;
	write_item(Y);
	unlock(Y);

write_lock(X);
read_item(X);
$X := X + Y$;
write_item(X);
unlock(X);	

Time

Result of schedule S: X=50, Y=50 (nonserializable)

Figure 22.3
Transactions that do not obey two-phase locking. (a) Two transactions T_1 and T_2. (b) Results of possible serial schedules of T_1 and T_2. (c) A nonserializable schedule S that uses locks.

shrinking phase. Hence, a read_lock(X) operation that downgrades an already held write lock on X can appear only in the shrinking phase.

Transactions T_1 and T_2 in Figure 22.3(a) do not follow the two-phase locking protocol because the write_lock(X) operation follows the unlock(Y) operation in T_1, and similarly the write_lock(Y) operation follows the unlock(X) operation in T_2. If we enforce two-phase locking, the transactions can be rewritten as T_1' and T_2', as shown in Figure 22.4. Now, the schedule shown in Figure 22.3(c) is not permitted for T_1' and T_2' (with their modified order of locking and unlocking operations) under the rules of locking described in Section 22.1.1 because T_1' will issue its write_lock(X) *before* it unlocks item Y; consequently, when T_2' issues its read_lock(X), it is forced to wait until T_1' releases the lock by issuing an unlock (X) in the schedule.

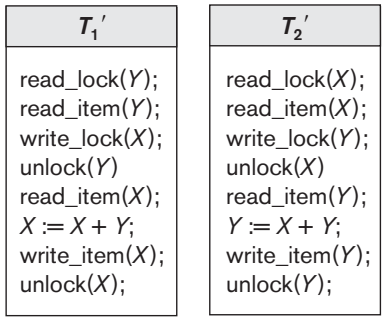


Figure 22.4
Transactions T_1' and T_2', which are the same as T_1 and T_2 in Figure 22.3, but follow the two-phase locking protocol. Note that they can produce a deadlock.

It can be proved that, if *every* transaction in a schedule follows the two-phase locking protocol, the schedule is *guaranteed to be serializable*, obviating the need to test for serializability of schedules. The locking protocol, by enforcing two-phase locking rules, also enforces serializability.

Two-phase locking may limit the amount of concurrency that can occur in a schedule because a transaction T may not be able to release an item X after it is through using it if T must lock an additional item Y later; or conversely, T must lock the additional item Y before it needs it so that it can release X. Hence, X must remain locked by T until all items that the transaction needs to read or write have been locked; only then can X be released by T. Meanwhile, another transaction seeking to access X may be forced to wait, even though T is done with X; conversely, if Y is locked earlier than it is needed, another transaction seeking to access Y is forced to wait even though T is not using Y yet. This is the price for guaranteeing serializability of all schedules without having to check the schedules themselves.

Although the two-phase locking protocol guarantees serializability (that is, every schedule that is permitted is serializable), it does not permit *all possible* serializable schedules (that is, some serializable schedules will be prohibited by the protocol).

Basic, Conservative, Strict, and Rigorous Two-Phase Locking. There are a number of variations of two-phase locking (2PL). The technique just described is known as **basic 2PL**. A variation known as **conservative 2PL** (or **static 2PL**) requires a transaction to lock all the items it accesses *before the transaction begins execution*, by **predeclaring** its *read-set* and *write-set*. Recall from Section 21.1.2 that the **read-set** of a transaction is the set of all items that the transaction reads, and the **write-set** is the set of all items that it writes. If any of the predeclared items needed cannot be locked, the transaction does not lock any item; instead, it waits until all the items are available for locking. Conservative 2PL is a deadlock-free protocol, as we will see in Section 22.1.3 when we discuss the deadlock problem. However, it is difficult to use in practice because of the need to predeclare the read-set and write-set, which is not possible in many situations.

In practice, the most popular variation of 2PL is **strict 2PL**, which guarantees strict schedules (see Section 21.4). In this variation, a transaction T does not release any of

its exclusive (write) locks until *after* it commits or aborts. Hence, no other transaction can read or write an item that is written by T unless T has committed, leading to a strict schedule for recoverability. Strict 2PL is not deadlock-free. A more restrictive variation of strict 2PL is **rigorous 2PL**, which also guarantees strict schedules. In this variation, a transaction T does not release any of its locks (exclusive or shared) until after it commits or aborts, and so it is easier to implement than strict 2PL. Notice the difference between conservative and rigorous 2PL: the former must lock all its items *before it starts*, so once the transaction starts it is in its shrinking phase; the latter does not unlock any of its items until *after it terminates* (by committing or aborting), so the transaction is in its expanding phase until it ends.

In many cases, the **concurrency control subsystem** itself is responsible for generating the read_lock and write_lock requests. For example, suppose the system is to enforce the strict 2PL protocol. Then, whenever transaction T issues a read_item(X), the system calls the read_lock(X) operation on behalf of T. If the state of LOCK(X) is write_locked by some other transaction T', the system places T in the waiting queue for item X; otherwise, it grants the read_lock(X) request and permits the read_item(X) operation of T to execute. On the other hand, if transaction T issues a write_item(X), the system calls the write_lock(X) operation on behalf of T. If the state of LOCK(X) is write_locked or read_locked by some other transaction T', the system places T in the waiting queue for item X; if the state of LOCK(X) is read_locked and T itself is the only transaction holding the read lock on X, the system upgrades the lock to write_locked and permits the write_item(X) operation by T. Finally, if the state of LOCK(X) is unlocked, the system grants the write_lock(X) request and permits the write_item(X) operation to execute. After each action, the system must update its lock table appropriately.

The use of locks can cause two additional problems: deadlock and starvation. We discuss these problems and their solutions in the next section.

22.1.3 Dealing with Deadlock and Starvation

Deadlock occurs when *each* transaction T in a set of *two or more transactions* is waiting for some item that is locked by some other transaction T' in the set. Hence, each transaction in the set is in a waiting queue, waiting for one of the other transactions in the set to release the lock on an item. But because the other transaction is also waiting, it will never release the lock. A simple example is shown in Figure 22.5(a), where the two transactions T_1' and T_2' are deadlocked in a partial schedule; T_1' is in the waiting queue for X, which is locked by T_2', while T_2' is in the waiting queue for Y, which is locked by T_1'. Meanwhile, neither T_1' nor T_2' nor any other transaction can access items X and Y.

Deadlock Prevention Protocols. One way to prevent deadlock is to use a **deadlock prevention protocol**.[5] One deadlock prevention protocol, which is used

[5]These protocols are not generally used in practice, either because of unrealistic assumptions or because of their possible overhead. Deadlock detection and timeouts (covered in the following sections) are more practical.

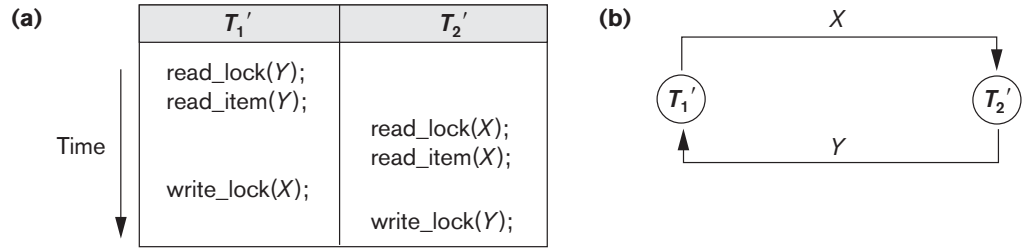


Figure 22.5
Illustrating the deadlock problem. (a) A partial schedule of T_1' and T_2' that is in a state of deadlock. (b) A wait-for graph for the partial schedule in (a).

in conservative two-phase locking, requires that every transaction lock *all the items it needs in advance* (which is generally not a practical assumption)—if any of the items cannot be obtained, none of the items are locked. Rather, the transaction waits and then tries again to lock all the items it needs. Obviously this solution further limits concurrency. A second protocol, which also limits concurrency, involves *ordering all the items* in the database and making sure that a transaction that needs several items will lock them according to that order. This requires that the programmer (or the system) is aware of the chosen order of the items, which is also not practical in the database context.

A number of other deadlock prevention schemes have been proposed that make a decision about what to do with a transaction involved in a possible deadlock situation: Should it be blocked and made to wait or should it be aborted, or should the transaction preempt and abort another transaction? Some of these techniques use the concept of **transaction timestamp** TS(T), which is a unique identifier assigned to each transaction. The timestamps are typically based on the order in which transactions are started; hence, if transaction T_1 starts before transaction T_2, then TS(T_1) < TS(T_2). Notice that the *older* transaction (which starts first) has the *smaller* timestamp value. Two schemes that prevent deadlock are called *wait-die* and *wound-wait*. Suppose that transaction T_i tries to lock an item X but is not able to because X is locked by some other transaction T_j with a conflicting lock. The rules followed by these schemes are:

- **Wait-die.** If TS(T_i) < TS(T_j), then (T_i older than T_j) T_i is allowed to wait; otherwise (T_i younger than T_j) abort T_i (T_i *dies*) and restart it later *with the same timestamp.*
- **Wound-wait.** If TS(T_i) < TS(T_j), then (T_i older than T_j) abort T_j (T_i *wounds* T_j) and restart it later *with the same timestamp;* otherwise (T_i younger than T_j) T_i is allowed to wait.

In wait-die, an older transaction is allowed to *wait for a younger transaction*, whereas a younger transaction requesting an item held by an older transaction is aborted and restarted. The wound-wait approach does the opposite: A younger transaction is allowed to *wait for an older one*, whereas an older transaction requesting an item

held by a younger transaction *preempts* the younger transaction by aborting it. Both schemes end up aborting the *younger* of the two transactions (the transaction that started later) that *may be involved* in a deadlock, assuming that this will waste less processing. It can be shown that these two techniques are *deadlock-free*, since in wait-die, transactions only wait for younger transactions so no cycle is created. Similarly, in wound-wait, transactions only wait for older transactions so no cycle is created. However, both techniques may cause some transactions to be aborted and restarted needlessly, even though those transactions may *never actually cause a deadlock*.

Another group of protocols that prevent deadlock do not require timestamps. These include the no waiting (NW) and cautious waiting (CW) algorithms. In the **no waiting algorithm**, if a transaction is unable to obtain a lock, it is immediately aborted and then restarted after a certain time delay without checking whether a deadlock will actually occur or not. In this case, no transaction ever waits, so no deadlock will occur. However, this scheme can cause transactions to abort and restart needlessly. The **cautious waiting** algorithm was proposed to try to reduce the number of needless aborts/restarts. Suppose that transaction T_i tries to lock an item X but is not able to do so because X is locked by some other transaction T_j with a conflicting lock. The cautious waiting rules are as follows:

- **Cautious waiting.** If T_j is not blocked (not waiting for some other locked item), then T_i is blocked and allowed to wait; otherwise abort T_i.

It can be shown that cautious waiting is deadlock-free, because no transaction will ever wait for another blocked transaction. By considering the time $b(T)$ at which each blocked transaction T was blocked, if the two transactions T_i and T_j above both become blocked, and T_i is waiting for T_j, then $b(T_i) < b(T_j)$, since T_i can only wait for T_j at a time when T_j is not blocked itself. Hence, the blocking times form a total ordering on all blocked transactions, so no cycle that causes deadlock can occur.

Deadlock Detection. A second, more practical approach to dealing with deadlock is **deadlock detection**, where the system checks if a state of deadlock actually exists. This solution is attractive if we know there will be little interference among the transactions—that is, if different transactions will rarely access the same items at the same time. This can happen if the transactions are short and each transaction locks only a few items, or if the transaction load is light. On the other hand, if transactions are long and each transaction uses many items, or if the transaction load is quite heavy, it may be advantageous to use a deadlock prevention scheme.

A simple way to detect a state of deadlock is for the system to construct and maintain a **wait-for graph**. One node is created in the wait-for graph for each transaction that is currently executing. Whenever a transaction T_i is waiting to lock an item X that is currently locked by a transaction T_j, a directed edge $(T_i \rightarrow T_j)$ is created in the wait-for graph. When T_j releases the lock(s) on the items that T_i was waiting for, the directed edge is dropped from the wait-for graph. We have a state of deadlock if and only if the wait-for graph has a cycle. One problem with this approach is the matter of determining *when* the system should check for a deadlock. One possi-

bility is to check for a cycle every time an edge is added to the wait-for graph, but this may cause excessive overhead. Criteria such as the number of currently executing transactions or the period of time several transactions have been waiting to lock items may be used instead to check for a cycle. Figure 22.5(b) shows the wait-for graph for the (partial) schedule shown in Figure 22.5(a).

If the system is in a state of deadlock, some of the transactions causing the deadlock must be aborted. Choosing which transactions to abort is known as **victim selection**. The algorithm for victim selection should generally avoid selecting transactions that have been running for a long time and that have performed many updates, and it should try instead to select transactions that have not made many changes (younger transactions).

Timeouts. Another simple scheme to deal with deadlock is the use of **timeouts**. This method is practical because of its low overhead and simplicity. In this method, if a transaction waits for a period longer than a system-defined timeout period, the system assumes that the transaction may be deadlocked and aborts it—regardless of whether a deadlock actually exists or not.

Starvation. Another problem that may occur when we use locking is **starvation**, which occurs when a transaction cannot proceed for an indefinite period of time while other transactions in the system continue normally. This may occur if the waiting scheme for locked items is unfair, giving priority to some transactions over others. One solution for starvation is to have a fair waiting scheme, such as using a **first-come-first-served** queue; transactions are enabled to lock an item in the order in which they originally requested the lock. Another scheme allows some transactions to have priority over others but increases the priority of a transaction the longer it waits, until it eventually gets the highest priority and proceeds. Starvation can also occur because of victim selection if the algorithm selects the same transaction as victim repeatedly, thus causing it to abort and never finish execution. The algorithm can use higher priorities for transactions that have been aborted multiple times to avoid this problem. The wait-die and wound-wait schemes discussed previously avoid starvation, because they restart a transaction that has been aborted with its same original timestamp, so the possibility that the same transaction is aborted repeatedly is slim.

22.2 Concurrency Control Based on Timestamp Ordering

The use of locks, combined with the 2PL protocol, guarantees serializability of schedules. The serializable schedules produced by 2PL have their equivalent serial schedules based on the order in which executing transactions lock the items they acquire. If a transaction needs an item that is already locked, it may be forced to wait until the item is released. Some transactions may be aborted and restarted because of the deadlock problem. A different approach that guarantees serializability involves using transaction timestamps to order transaction execution for an equiva-

lent serial schedule. In Section 22.2.1 we discuss timestamps, and in Section 22.2.2 we discuss how serializability is enforced by ordering transactions based on their timestamps.

22.2.1 Timestamps

Recall that a **timestamp** is a unique identifier created by the DBMS to identify a transaction. Typically, timestamp values are assigned in the order in which the transactions are submitted to the system, so a timestamp can be thought of as the *transaction start time*. We will refer to the timestamp of transaction T as **TS(*T*)**. Concurrency control techniques based on timestamp ordering do not use locks; hence, *deadlocks cannot occur*.

Timestamps can be generated in several ways. One possibility is to use a counter that is incremented each time its value is assigned to a transaction. The transaction timestamps are numbered 1, 2, 3, ... in this scheme. A computer counter has a finite maximum value, so the system must periodically reset the counter to zero when no transactions are executing for some short period of time. Another way to implement timestamps is to use the current date/time value of the system clock and ensure that no two timestamp values are generated during the same tick of the clock.

22.2.2 The Timestamp Ordering Algorithm

The idea for this scheme is to order the transactions based on their timestamps. A schedule in which the transactions participate is then serializable, and the *only equivalent serial schedule permitted* has the transactions in order of their timestamp values. This is called **timestamp ordering** (**TO**). Notice how this differs from 2PL, where a schedule is serializable by being equivalent to some serial schedule allowed by the locking protocols. In timestamp ordering, however, the schedule is equivalent to the *particular serial order* corresponding to the order of the transaction timestamps. The algorithm must ensure that, for each item accessed by *conflicting operations* in the schedule, the order in which the item is accessed does not violate the timestamp order. To do this, the algorithm associates with each database item X two timestamp (**TS**) values:

1. **read_TS(*X*).** The **read timestamp** of item X is the largest timestamp among all the timestamps of transactions that have successfully read item X—that is, read_TS(X) = TS(T), where T is the *youngest* transaction that has read X successfully.
2. **write_TS(*X*).** The **write timestamp** of item X is the largest of all the timestamps of transactions that have successfully written item X—that is, write_TS(X) = TS(T), where T is the *youngest* transaction that has written X successfully.

Basic Timestamp Ordering (TO). Whenever some transaction T tries to issue a read_item(X) or a write_item(X) operation, the **basic TO** algorithm compares the timestamp of T with read_TS(X) and write_TS(X) to ensure that the timestamp

order of transaction execution is not violated. If this order is violated, then transaction T is aborted and resubmitted to the system as a new transaction with a *new timestamp*. If T is aborted and rolled back, any transaction T_1 that may have used a value written by T must also be rolled back. Similarly, any transaction T_2 that may have used a value written by T_1 must also be rolled back, and so on. This effect is known as **cascading rollback** and is one of the problems associated with basic TO, since the schedules produced are not guaranteed to be recoverable. An *additional protocol* must be enforced to ensure that the schedules are recoverable, cascadeless, or strict. We first describe the basic TO algorithm here. The concurrency control algorithm must check whether conflicting operations violate the timestamp ordering in the following two cases:

1. Whenever a transaction T issues a write_item(X) operation, the following is checked:
   a. If read_TS(X) > TS(T) or if write_TS(X) > TS(T), then abort and roll back T and reject the operation. This should be done because some *younger* transaction with a timestamp greater than TS(T)—and hence *after* T in the timestamp ordering—has already read or written the value of item X before T had a chance to write X, thus violating the timestamp ordering.
   b. If the condition in part (a) does not occur, then execute the write_item(X) operation of T and set write_TS(X) to TS(T).
2. Whenever a transaction T issues a read_item(X) operation, the following is checked:
   a. If write_TS(X) > TS(T), then abort and roll back T and reject the operation. This should be done because some younger transaction with timestamp greater than TS(T)—and hence *after* T in the timestamp ordering—has already written the value of item X before T had a chance to read X.
   b. If write_TS(X) ≤ TS(T), then execute the read_item(X) operation of T and set read_TS(X) to the *larger* of TS(T) and the current read_TS(X).

Whenever the basic TO algorithm detects two *conflicting operations* that occur in the incorrect order, it rejects the later of the two operations by aborting the transaction that issued it. The schedules produced by basic TO are hence guaranteed to be *conflict serializable*, like the 2PL protocol. However, some schedules are possible under each protocol that are not allowed under the other. Thus, *neither* protocol allows *all possible* serializable schedules. As mentioned earlier, deadlock does not occur with timestamp ordering. However, cyclic restart (and hence starvation) may occur if a transaction is continually aborted and restarted.

Strict Timestamp Ordering (TO). A variation of basic TO called **strict TO** ensures that the schedules are both **strict** (for easy recoverability) and (conflict) serializable. In this variation, a transaction T that issues a read_item(X) or write_item(X) such that TS(T) > write_TS(X) has its read or write operation *delayed* until the transaction T' that *wrote* the value of X (hence TS(T') = write_TS(X)) has committed or aborted. To implement this algorithm, it is necessary to simulate the

locking of an item X that has been written by transaction T' until T' is either committed or aborted. This algorithm *does not cause deadlock*, since T waits for T' only if $\mathsf{TS}(T) > \mathsf{TS}(T')$.

Thomas's Write Rule. A modification of the basic TO algorithm, known as **Thomas's write rule**, does not enforce conflict serializability, but it rejects fewer write operations by modifying the checks for the write_item(X) operation as follows:

1. If read_TS(X) > TS(T), then abort and roll back T and reject the operation.
2. If write_TS(X) > TS(T), then do not execute the write operation but continue processing. This is because some transaction with timestamp greater than TS(T)—and hence after T in the timestamp ordering—has already written the value of X. Thus, we must ignore the write_item(X) operation of T because it is already outdated and obsolete. Notice that any conflict arising from this situation would be detected by case (1).
3. If neither the condition in part (1) nor the condition in part (2) occurs, then execute the write_item(X) operation of T and set write_TS(X) to TS(T).

22.3 Multiversion Concurrency Control Techniques

Other protocols for concurrency control keep the old values of a data item when the item is updated. These are known as **multiversion concurrency control**, because several versions (values) of an item are maintained. When a transaction requires access to an item, an *appropriate* version is chosen to maintain the serializability of the currently executing schedule, if possible. The idea is that some read operations that would be rejected in other techniques can still be accepted by reading an *older version* of the item to maintain serializability. When a transaction writes an item, it writes a *new version* and the old version(s) of the item are retained. Some multiversion concurrency control algorithms use the concept of view serializability rather than conflict serializability.

An obvious drawback of multiversion techniques is that more storage is needed to maintain multiple versions of the database items. However, older versions may have to be maintained anyway—for example, for recovery purposes. In addition, some database applications require older versions to be kept to maintain a history of the evolution of data item values. The extreme case is a *temporal database* (see Secton 26.2), which keeps track of all changes and the times at which they occurred. In such cases, there is no additional storage penalty for multiversion techniques, since older versions are already maintained.

Several multiversion concurrency control schemes have been proposed. We discuss two schemes here, one based on timestamp ordering and the other based on 2PL. In addition, the validation concurrency control method (see Section 22.4) also maintains multiple versions.

22.3.1 Multiversion Technique Based on Timestamp Ordering

In this method, several versions X_1, X_2, ..., X_k of each data item X are maintained. For *each version*, the value of version X_i and the following two timestamps are kept:

1. **read_TS(X_i).** The **read timestamp** of X_i is the largest of all the timestamps of transactions that have successfully read version X_i.
2. **write_TS(X_i).** The **write timestamp** of X_i is the timestamp of the transaction that wrote the value of version X_i.

Whenever a transaction T is allowed to execute a write_item(X) operation, a new version X_{k+1} of item X is created, with both the write_TS(X_{k+1}) and the read_TS(X_{k+1}) set to TS(T). Correspondingly, when a transaction T is allowed to read the value of version X_i, the value of read_TS(X_i) is set to the larger of the current read_TS(X_i) and TS(T).

To ensure serializability, the following rules are used:

1. If transaction T issues a write_item(X) operation, and version i of X has the highest write_TS(X_i) of all versions of X that is also *less than or equal to* TS(T), and read_TS(X_i) > TS(T), then abort and roll back transaction T; otherwise, create a new version X_j of X with read_TS(X_j) = write_TS(X_j) = TS(T).
2. If transaction T issues a read_item(X) operation, find the version i of X that has the highest write_TS(X_i) of all versions of X that is also *less than or equal to* TS(T); then return the value of X_i to transaction T, and set the value of read_TS(X_i) to the larger of TS(T) and the current read_TS(X_i).

As we can see in case 2, a read_item(X) is always successful, since it finds the appropriate version X_i to read based on the write_TS of the various existing versions of X. In case 1, however, transaction T may be aborted and rolled back. This happens if T attempts to write a version of X that should have been read by another transaction T' whose timestamp is read_TS(X_i); however, T' has already read version X_i, which was written by the transaction with timestamp equal to write_TS(X_i). If this conflict occurs, T is rolled back; otherwise, a new version of X, written by transaction T, is created. Notice that if T is rolled back, cascading rollback may occur. Hence, to ensure recoverability, a transaction T should not be allowed to commit until after all the transactions that have written some version that T has read have committed.

22.3.2 Multiversion Two-Phase Locking Using Certify Locks

In this multiple-mode locking scheme, there are *three locking modes* for an item: read, write, and *certify*, instead of just the two modes (read, write) discussed previously. Hence, the state of LOCK(X) for an item X can be one of read-locked, write-locked, certify-locked, or unlocked. In the standard locking scheme, with only read and write locks (see Section 22.1.1), a write lock is an exclusive lock. We can describe the relationship between read and write locks in the standard scheme by means of the **lock compatibility table** shown in Figure 22.6(a). An entry of *Yes* means that if a transaction T holds the type of lock specified in the column header

(a)

	Read	Write
Read	Yes	No
Write	No	No

(b)

	Read	Write	Certify
Read	Yes	Yes	No
Write	Yes	No	No
Certify	No	No	No

Figure 22.6
Lock compatibility tables. (a) A compatibility table for read/write locking scheme. (b) A compatibility table for read/write/certify locking scheme.

on item X and if transaction T' requests the type of lock specified in the row header on the same item X, then T' *can obtain the lock* because the locking modes are compatible. On the other hand, an entry of *No* in the table indicates that the locks are not compatible, so T' *must wait* until T *releases* the lock.

In the standard locking scheme, once a transaction obtains a write lock on an item, no other transactions can access that item. The idea behind multiversion 2PL is to allow other transactions T' to read an item X while a single transaction T holds a write lock on X. This is accomplished by allowing *two versions* for each item X; one version must always have been written by some committed transaction. The second version X' is created when a transaction T acquires a write lock on the item. Other transactions can continue to read the *committed version* of X while T holds the write lock. Transaction T can write the value of X' as needed, without affecting the value of the committed version X. However, once T is ready to commit, it must obtain a **certify lock** on all items that it currently holds write locks on before it can commit. The certify lock is not compatible with read locks, so the transaction may have to delay its commit until all its write-locked items are released by any reading transactions in order to obtain the certify locks. Once the certify locks—which are exclusive locks—are acquired, the committed version X of the data item is set to the value of version X', version X' is discarded, and the certify locks are then released. The lock compatibility table for this scheme is shown in Figure 22.6(b).

In this multiversion 2PL scheme, reads can proceed concurrently with a single write operation—an arrangement not permitted under the standard 2PL schemes. The cost is that a transaction may have to delay its commit until it obtains exclusive certify locks on *all the items* it has updated. It can be shown that this scheme avoids cascading aborts, since transactions are only allowed to read the version X that was written by a committed transaction. However, deadlocks may occur if upgrading of a read lock to a write lock is allowed, and these must be handled by variations of the techniques discussed in Section 22.1.3.

22.4 Validation (Optimistic) Concurrency Control Techniques

In all the concurrency control techniques we have discussed so far, a certain degree of checking is done *before* a database operation can be executed. For example, in locking, a check is done to determine whether the item being accessed is locked. In timestamp ordering, the transaction timestamp is checked against the read and write timestamps of the item. Such checking represents overhead during transaction execution, with the effect of slowing down the transactions.

In **optimistic concurrency control techniques**, also known as **validation** or **certification techniques**, *no checking* is done while the transaction is executing. Several theoretical concurrency control methods are based on the validation technique. We will describe only one scheme here. In this scheme, updates in the transaction are *not* applied directly to the database items until the transaction reaches its end. During transaction execution, all updates are applied to *local copies* of the data items that are kept for the transaction.[6] At the end of transaction execution, a **validation phase** checks whether any of the transaction's updates violate serializability. Certain information needed by the validation phase must be kept by the system. If serializability is not violated, the transaction is committed and the database is updated from the local copies; otherwise, the transaction is aborted and then restarted later.

There are three phases for this concurrency control protocol:

1. **Read phase.** A transaction can read values of committed data items from the database. However, updates are applied only to local copies (versions) of the data items kept in the transaction workspace.
2. **Validation phase.** Checking is performed to ensure that serializability will not be violated if the transaction updates are applied to the database.
3. **Write phase.** If the validation phase is successful, the transaction updates are applied to the database; otherwise, the updates are discarded and the transaction is restarted.

The idea behind optimistic concurrency control is to do all the checks at once; hence, transaction execution proceeds with a minimum of overhead until the validation phase is reached. If there is little interference among transactions, most will be validated successfully. However, if there is much interference, many transactions that execute to completion will have their results discarded and must be restarted later. Under these circumstances, optimistic techniques do not work well. The techniques are called *optimistic* because they assume that little interference will occur and hence that there is no need to do checking during transaction execution.

The optimistic protocol we describe uses transaction timestamps and also requires that the write_sets and read_sets of the transactions be kept by the system. Additionally, *start* and *end* times for some of the three phases need to be kept for

[6]Note that this can be considered as keeping multiple versions of items!

each transaction. Recall that the write_set of a transaction is the set of items it writes, and the read_set is the set of items it reads. In the validation phase for transaction T_i, the protocol checks that T_i does not interfere with any committed transactions or with any other transactions currently in their validation phase. The validation phase for T_i checks that, for *each* such transaction T_j that is either committed or is in its validation phase, *one* of the following conditions holds:

1. Transaction T_j completes its write phase before T_i starts its read phase.
2. T_i starts its write phase after T_j completes its write phase, and the read_set of T_i has no items in common with the write_set of T_j.
3. Both the read_set and write_set of T_i have no items in common with the write_set of T_j, and T_j completes its read phase before T_i completes its read phase.

When validating transaction T_i, the first condition is checked first for each transaction T_j, since (1) is the simplest condition to check. Only if condition 1 is false is condition 2 checked, and only if (2) is false is condition 3—the most complex to evaluate—checked. If any one of these three conditions holds, there is no interference and T_i is validated successfully. If *none* of these three conditions holds, the validation of transaction T_i fails and it is aborted and restarted later because interference *may* have occurred.

22.5 Granularity of Data Items and Multiple Granularity Locking

All concurrency control techniques assume that the database is formed of a number of named data items. A database item could be chosen to be one of the following:

- A database record
- A field value of a database record
- A disk block
- A whole file
- The whole database

The granularity can affect the performance of concurrency control and recovery. In Section 22.5.1, we discuss some of the tradeoffs with regard to choosing the granularity level used for locking, and in Section 22.5.2 we discuss a multiple granularity locking scheme, where the granularity level (size of the data item) may be changed dynamically.

22.5.1 Granularity Level Considerations for Locking

The size of data items is often called the **data item granularity**. *Fine granularity* refers to small item sizes, whereas *coarse granularity* refers to large item sizes. Several tradeoffs must be considered in choosing the data item size. We will discuss data item size in the context of locking, although similar arguments can be made for other concurrency control techniques.

First, notice that the larger the data item size is, the lower the degree of concurrency permitted. For example, if the data item size is a disk block, a transaction T that needs to lock a record B must lock the whole disk block X that contains B because a lock is associated with the whole data item (block). Now, if another transaction S wants to lock a different record C that happens to reside in the same block X in a conflicting lock mode, it is forced to wait. If the data item size was a single record, transaction S would be able to proceed, because it would be locking a different data item (record).

On the other hand, the smaller the data item size is, the more the number of items in the database. Because every item is associated with a lock, the system will have a larger number of active locks to be handled by the lock manager. More lock and unlock operations will be performed, causing a higher overhead. In addition, more storage space will be required for the lock table. For timestamps, storage is required for the read_TS and write_TS for each data item, and there will be similar overhead for handling a large number of items.

Given the above tradeoffs, an obvious question can be asked: What is the best item size? The answer is that it *depends on the types of transactions involved*. If a typical transaction accesses a small number of records, it is advantageous to have the data item granularity be one record. On the other hand, if a transaction typically accesses many records in the same file, it may be better to have block or file granularity so that the transaction will consider all those records as one (or a few) data items.

22.5.2 Multiple Granularity Level Locking

Since the best granularity size depends on the given transaction, it seems appropriate that a database system should support multiple levels of granularity, where the granularity level can be different for various mixes of transactions. Figure 22.7 shows a simple granularity hierarchy with a database containing two files, each file containing several disk pages, and each page containing several records. This can be used to illustrate a **multiple granularity level** 2PL protocol, where a lock can be requested at any level. However, additional types of locks will be needed to support such a protocol efficiently.

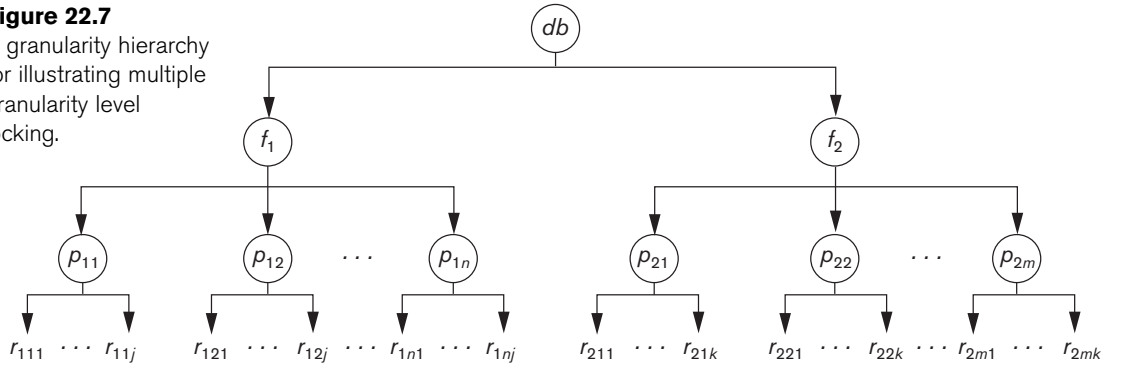


Figure 22.7
A granularity hierarchy for illustrating multiple granularity level locking.

Consider the following scenario, with only shared and exclusive lock types, that refers to the example in Figure 22.7. Suppose transaction T_1 wants to update *all the records* in file f_1, and T_1 requests and is granted an exclusive lock for f_1. Then all of f_1's pages (p_{11} through p_{1n})—and the records contained on those pages—are locked in exclusive mode. This is beneficial for T_1 because setting a single file-level lock is more efficient than setting n page-level locks or having to lock each individual record. Now suppose another transaction T_2 only wants to read record r_{1nj} from page p_{1n} of file f_1; then T_2 would request a shared record-level lock on r_{1nj}. However, the database system (that is, the transaction manager or more specifically the lock manager) must verify the compatibility of the requested lock with already held locks. One way to verify this is to traverse the tree from the leaf r_{1nj} to p_{1n} to f_1 to *db*. If at any time a conflicting lock is held on any of those items, then the lock request for r_{1nj} is denied and T_2 is blocked and must wait. This traversal would be fairly efficient.

However, what if transaction T_2's request came *before* transaction T_1's request? In this case, the shared record lock is granted to T_2 for r_{1nj}, but when T_1's file-level lock is requested, it is quite difficult for the lock manager to check all nodes (pages and records) that are descendants of node f_1 for a lock conflict. This would be very inefficient and would defeat the purpose of having multiple granularity level locks.

To make multiple granularity level locking practical, additional types of locks, called **intention locks**, are needed. The idea behind intention locks is for a transaction to indicate, along the path from the root to the desired node, what type of lock (shared or exclusive) it will require from one of the node's descendants. There are three types of intention locks:

1. Intention-shared (IS) indicates that one or more shared locks will be requested on some descendant node(s).
2. Intention-exclusive (IX) indicates that one or more exclusive locks will be requested on some descendant node(s).
3. Shared-intention-exclusive (SIX) indicates that the current node is locked in shared mode but that one or more exclusive locks will be requested on some descendant node(s).

The compatibility table of the three intention locks, and the shared and exclusive locks, is shown in Figure 22.8. Besides the introduction of the three types of intention locks, an appropriate locking protocol must be used. The **multiple granularity locking (MGL)** protocol consists of the following rules:

1. The lock compatibility (based on Figure 22.8) must be adhered to.
2. The root of the tree must be locked first, in any mode.
3. A node N can be locked by a transaction T in S or IS mode only if the parent node N is already locked by transaction T in either IS or IX mode.
4. A node N can be locked by a transaction T in X, IX, or SIX mode only if the parent of node N is already locked by transaction T in either IX or SIX mode.
5. A transaction T can lock a node only if it has not unlocked any node (to enforce the 2PL protocol).

	IS	IX	S	SIX	X
IS	Yes	Yes	Yes	Yes	No
IX	Yes	Yes	No	No	No
S	Yes	No	Yes	No	No
SIX	Yes	No	No	No	No
X	No	No	No	No	No

Figure 22.8
Lock compatibility matrix for multiple granularity locking.

6. A transaction T can unlock a node, N, only if none of the children of node N are currently locked by T.

Rule 1 simply states that conflicting locks cannot be granted. Rules 2, 3, and 4 state the conditions when a transaction may lock a given node in any of the lock modes. Rules 5 and 6 of the MGL protocol enforce 2PL rules to produce serializable schedules. To illustrate the MGL protocol with the database hierarchy in Figure 22.7, consider the following three transactions:

1. T_1 wants to update record r_{111} and record r_{211}.
2. T_2 wants to update all records on page p_{12}.
3. T_3 wants to read record r_{11j} and the entire f_2 file.

Figure 22.9 shows a possible serializable schedule for these three transactions. Only the lock and unlock operations are shown. The notation <lock_type>(<item>) is used to display the locking operations in the schedule.

The multiple granularity level protocol is especially suited when processing a mix of transactions that include (1) short transactions that access only a few items (records or fields) and (2) long transactions that access entire files. In this environment, less transaction blocking and less locking overhead is incurred by such a protocol when compared to a single level granularity locking approach.

22.6 Using Locks for Concurrency Control in Indexes

Two-phase locking can also be applied to indexes (see Chapter 18), where the nodes of an index correspond to disk pages. However, holding locks on index pages until the shrinking phase of 2PL could cause an undue amount of transaction blocking because searching an index always *starts at the root*. Therefore, if a transaction wants to insert a record (write operation), the root would be locked in exclusive mode, so all other conflicting lock requests for the index must wait until the transaction enters its shrinking phase. This blocks all other transactions from accessing the index, so in practice other approaches to locking an index must be used.

T_1	T_2	T_3
IX(*db*)		
IX(f_1)		
	IX(*db*)	
		IS(*db*)
		IS(f_1)
		IS(p_{11})
IX(p_{11})		
X(r_{111})		
	IX(f_1)	
	X(p_{12})	
		S(r_{11j})
IX(f_2)		
IX(p_{21})		
X(p_{211})		
unlock(r_{211})		
unlock(p_{21})		
unlock(f_2)		
		S(f_2)
	unlock(p_{12})	
	unlock(f_1)	
	unlock(*db*)	
unlock(r_{111})		
unlock(p_{11})		
unlock(f_1)		
unlock(*db*)		
		unlock(r_{11j})
		unlock(p_{11})
		unlock(f_1)
		unlock(f_2)
		unlock(*db*)

Figure 22.9
Lock operations to illustrate a serializable schedule.

The tree structure of the index can be taken advantage of when developing a concurrency control scheme. For example, when an index search (read operation) is being executed, a path in the tree is traversed from the root to a leaf. Once a lower-level node in the path has been accessed, the higher-level nodes in that path will not be used again. So once a read lock on a child node is obtained, the lock on the parent can be released. When an insertion is being applied to a leaf node (that is, when a key and a pointer are inserted), then a specific leaf node must be locked in exclusive mode. However, if that node is not full, the insertion will not cause changes to higher-level index nodes, which implies that they need not be locked exclusively.

A conservative approach for insertions would be to lock the root node in exclusive mode and then to access the appropriate child node of the root. If the child node is

not full, then the lock on the root node can be released. This approach can be applied all the way down the tree to the leaf, which is typically three or four levels from the root. Although exclusive locks are held, they are soon released. An alternative, more **optimistic approach** would be to request and hold *shared* locks on the nodes leading to the leaf node, with an *exclusive* lock on the leaf. If the insertion causes the leaf to split, insertion will propagate to one or more higher-level nodes. Then, the locks on the higher-level nodes can be upgraded to exclusive mode.

Another approach to index locking is to use a variant of the B^+-tree, called the **B-link tree**. In a B-link tree, sibling nodes on the same level are linked at every level. This allows shared locks to be used when requesting a page and requires that the lock be released before accessing the child node. For an insert operation, the shared lock on a node would be upgraded to exclusive mode. If a split occurs, the parent node must be relocked in exclusive mode. One complication is for search operations executed concurrently with the update. Suppose that a concurrent update operation follows the same path as the search, and inserts a new entry into the leaf node. Additionally, suppose that the insert causes that leaf node to split. When the insert is done, the search process resumes, following the pointer to the desired leaf, only to find that the key it is looking for is not present because the split has moved that key into a new leaf node, which would be the *right sibling* of the original leaf node. However, the search process can still succeed if it follows the pointer (link) in the original leaf node to its right sibling, where the desired key has been moved.

Handling the deletion case, where two or more nodes from the index tree merge, is also part of the B-link tree concurrency protocol. In this case, locks on the nodes to be merged are held as well as a lock on the parent of the two nodes to be merged.

22.7 Other Concurrency Control Issues

In this section we discuss some other issues relevant to concurrency control. In Section 22.7.1, we discuss problems associated with insertion and deletion of records and the so-called *phantom problem*, which may occur when records are inserted. This problem was described as a potential problem requiring a concurrency control measure in Section 21.6. In Section 22.7.2 we discuss problems that may occur when a transaction outputs some data to a monitor before it commits, and then the transaction is later aborted.

22.7.1 Insertion, Deletion, and Phantom Records

When a new data item is **inserted** in the database, it obviously cannot be accessed until after the item is created and the insert operation is completed. In a locking environment, a lock for the item can be created and set to exclusive (write) mode; the lock can be released at the same time as other write locks would be released, based on the concurrency control protocol being used. For a timestamp-based protocol, the read and write timestamps of the new item are set to the timestamp of the creating transaction.

Next, consider a **deletion operation** that is applied on an existing data item. For locking protocols, again an exclusive (write) lock must be obtained before the transaction can delete the item. For timestamp ordering, the protocol must ensure that no later transaction has read or written the item before allowing the item to be deleted.

A situation known as the **phantom problem** can occur when a new record that is being inserted by some transaction T satisfies a condition that a set of records accessed by another transaction T' must satisfy. For example, suppose that transaction T is inserting a new EMPLOYEE record whose Dno = 5, while transaction T' is accessing all EMPLOYEE records whose Dno = 5 (say, to add up all their Salary values to calculate the personnel budget for department 5). If the equivalent serial order is T followed by T', then T' must read the new EMPLOYEE record and include its Salary in the sum calculation. For the equivalent serial order T' followed by T, the new salary should not be included. Notice that although the transactions logically conflict, in the latter case there is really no record (data item) in common between the two transactions, since T' may have locked all the records with Dno = 5 *before* T inserted the new record. This is because the record that causes the conflict is a **phantom record** that has suddenly appeared in the database on being inserted. If other operations in the two transactions conflict, the conflict due to the phantom record may not be recognized by the concurrency control protocol.

One solution to the phantom record problem is to use **index locking**, as discussed in Section 22.6. Recall from Chapter 18 that an index includes entries that have an attribute value, plus a set of pointers to all records in the file with that value. For example, an index on Dno of EMPLOYEE would include an entry for each distinct Dno value, plus a set of pointers to all EMPLOYEE records with that value. If the index entry is locked before the record itself can be accessed, then the conflict on the phantom record can be detected, because transaction T' would request a read lock on the *index entry* for Dno = 5, and T would request a write lock on the same entry *before* they could place the locks on the actual records. Since the index locks conflict, the phantom conflict would be detected.

A more general technique, called **predicate locking**, would lock access to all records that satisfy an arbitrary *predicate* (condition) in a similar manner; however, predicate locks have proved to be difficult to implement efficiently.

22.7.2 Interactive Transactions

Another problem occurs when interactive transactions read input and write output to an interactive device, such as a monitor screen, before they are committed. The problem is that a user can input a value of a data item to a transaction T that is based on some value written to the screen by transaction T', which may not have committed. This dependency between T and T' cannot be modeled by the system concurrency control method, since it is only based on the user interacting with the two transactions.

An approach to dealing with this problem is to postpone output of transactions to the screen until they have committed.

22.7.3 Latches

Locks held for a short duration are typically called **latches**. Latches do not follow the usual concurrency control protocol such as two-phase locking. For example, a latch can be used to guarantee the physical integrity of a page when that page is being written from the buffer to disk. A latch would be acquired for the page, the page written to disk, and then the latch released.

22.8 Summary

In this chapter we discussed DBMS techniques for concurrency control. We started by discussing lock-based protocols, which are by far the most commonly used in practice. We described the two-phase locking (2PL) protocol and a number of its variations: basic 2PL, strict 2PL, conservative 2PL, and rigorous 2PL. The strict and rigorous variations are more common because of their better recoverability properties. We introduced the concepts of shared (read) and exclusive (write) locks, and showed how locking can guarantee serializability when used in conjunction with the two-phase locking rule. We also presented various techniques for dealing with the deadlock problem, which can occur with locking. In practice, it is common to use timeouts and deadlock detection (wait-for graphs).

We presented other concurrency control protocols that are not used often in practice but are important for the theoretical alternatives they show for solving this problem. These include the timestamp ordering protocol, which ensures serializability based on the order of transaction timestamps. Timestamps are unique, system-generated transaction identifiers. We discussed Thomas's write rule, which improves performance but does not guarantee conflict serializability. The strict timestamp ordering protocol was also presented. We discussed two multiversion protocols, which assume that older versions of data items can be kept in the database. One technique, called multiversion two-phase locking (which has been used in practice), assumes that two versions can exist for an item and attempts to increase concurrency by making write and read locks compatible (at the cost of introducing an additional certify lock mode). We also presented a multiversion protocol based on timestamp ordering, and an example of an optimistic protocol, which is also known as a certification or validation protocol.

Then we turned our attention to the important practical issue of data item granularity. We described a multigranularity locking protocol that allows the change of granularity (item size) based on the current transaction mix, with the goal of improving the performance of concurrency control. An important practical issue was then presented, which is to develop locking protocols for indexes so that indexes do not become a hindrance to concurrent access. Finally, we introduced the phantom problem and problems with interactive transactions, and briefly described the concept of latches and how it differs from locks.

Review Questions

22.1. What is the two-phase locking protocol? How does it guarantee serializability?

22.2. What are some variations of the two-phase locking protocol? Why is strict or rigorous two-phase locking often preferred?

22.3. Discuss the problems of deadlock and starvation, and the different approaches to dealing with these problems.

22.4. Compare binary locks to exclusive/shared locks. Why is the latter type of locks preferable?

22.5. Describe the wait-die and wound-wait protocols for deadlock prevention.

22.6. Describe the cautious waiting, no waiting, and timeout protocols for deadlock prevention.

22.7. What is a timestamp? How does the system generate timestamps?

22.8. Discuss the timestamp ordering protocol for concurrency control. How does strict timestamp ordering differ from basic timestamp ordering?

22.9. Discuss two multiversion techniques for concurrency control.

22.10. What is a certify lock? What are the advantages and disadvantages of using certify locks?

22.11. How do optimistic concurrency control techniques differ from other concurrency control techniques? Why are they also called validation or certification techniques? Discuss the typical phases of an optimistic concurrency control method.

22.12. How does the granularity of data items affect the performance of concurrency control? What factors affect selection of granularity size for data items?

22.13. What type of lock is needed for insert and delete operations?

22.14. What is multiple granularity locking? Under what circumstances is it used?

22.15. What are intention locks?

22.16. When are latches used?

22.17. What is a phantom record? Discuss the problem that a phantom record can cause for concurrency control.

22.18. How does index locking resolve the phantom problem?

22.19. What is a predicate lock?

Exercises

22.20. Prove that the basic two-phase locking protocol guarantees conflict serializability of schedules. (*Hint*: Show that if a serializability graph for a schedule has a cycle, then at least one of the transactions participating in the schedule does not obey the two-phase locking protocol.)

22.21. Modify the data structures for multiple-mode locks and the algorithms for read_lock(X), write_lock(X), and unlock(X) so that upgrading and downgrading of locks are possible. (*Hint*: The lock needs to check the transaction id(s) that hold the lock, if any.)

22.22. Prove that strict two-phase locking guarantees strict schedules.

22.23. Prove that the wait-die and wound-wait protocols avoid deadlock and starvation.

22.24. Prove that cautious waiting avoids deadlock.

22.25. Apply the timestamp ordering algorithm to the schedules in Figure 21.8(b) and (c), and determine whether the algorithm will allow the execution of the schedules.

22.26. Repeat Exercise 22.25, but use the multiversion timestamp ordering method.

22.27. Why is two-phase locking not used as a concurrency control method for indexes such as B^+-trees?

22.28. The compatibility matrix in Figure 22.8 shows that IS and IX locks are compatible. Explain why this is valid.

22.29. The MGL protocol states that a transaction T can unlock a node N, only if none of the children of node N are still locked by transaction T. Show that without this condition, the MGL protocol would be incorrect.

Selected Bibliography

The two-phase locking protocol and the concept of predicate locks were first proposed by Eswaran et al. (1976). Bernstein et al. (1987), Gray and Reuter (1993), and Papadimitriou (1986) focus on concurrency control and recovery. Kumar (1996) focuses on performance of concurrency control methods. Locking is discussed in Gray et al. (1975), Lien and Weinberger (1978), Kedem and Silbershatz (1980), and Korth (1983). Deadlocks and wait-for graphs were formalized by Holt (1972), and the wait-wound and wound-die schemes are presented in Rosenkrantz et al. (1978). Cautious waiting is discussed in Hsu and Zhang (1992). Helal et al. (1993) compares various locking approaches. Timestamp-based concurrency control techniques are discussed in Bernstein and Goodman (1980) and Reed (1983). Optimistic concurrency control is discussed in Kung and Robinson (1981) and Bassiouni (1988). Papadimitriou and Kanellakis (1979) and Bernstein and

Goodman (1983) discuss multiversion techniques. Multiversion timestamp ordering was proposed in Reed (1979, 1983), and multiversion two-phase locking is discussed in Lai and Wilkinson (1984). A method for multiple locking granularities was proposed in Gray et al. (1975), and the effects of locking granularities are analyzed in Ries and Stonebraker (1977). Bhargava and Reidl (1988) presents an approach for dynamically choosing among various concurrency control and recovery methods. Concurrency control methods for indexes are presented in Lehman and Yao (1981) and in Shasha and Goodman (1988). A performance study of various B^+-tree concurrency control algorithms is presented in Srinivasan and Carey (1991).

Other work on concurrency control includes semantic-based concurrency control (Badrinath and Ramamritham, 1992), transaction models for long-running activities (Dayal et al., 1991), and multilevel transaction management (Hasse and Weikum, 1991).

chapter 23

Database Recovery Techniques

In this chapter we discuss some of the techniques that can be used for database recovery from failures. In Section 21.1.4 we discussed the different causes of failure, such as system crashes and transaction errors. Also, in Section 21.2, we covered many of the concepts that are used by recovery processes, such as the system log and commit points.

This chapter presents additional concepts that are relevant to recovery protocols, and provides an overview of the various database recovery algorithms We start in Section 23.1 with an outline of a typical recovery procedure and a categorization of recovery algorithms, and then we discuss several recovery concepts, including write-ahead logging, in-place versus shadow updates, and the process of rolling back (undoing) the effect of an incomplete or failed transaction. In Section 23.2 we present recovery techniques based on *deferred update,* also known as the NO-UNDO/REDO technique, where the data on disk is not updated until *after* a transaction commits. In Section 23.3 we discuss recovery techniques based on *immediate update*, where data can be updated on disk during transaction execution; 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 23.4. An example of a practical DBMS recovery scheme, called ARIES, is presented in Section 23.5. Recovery in multidatabases is briefly discussed in Section 23.6. Finally, techniques for recovery from catastrophic failure are discussed in Section 23.7. Section 23.8 summarizes the chapter.

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 at the end of the chapter and the online and printed 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 methods. We will 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.

23.1 Recovery Concepts

23.1.1 Recovery Outline and Categorization of Recovery Algorithms

Recovery from transaction failures usually means that the database is *restored* to the most recent consistent state just before the time of failure. To do this, the system must keep information about the changes that were applied to data items by the various transactions. This information is typically kept in the **system log**, as we discussed in Section 21.2.2. 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 *backed up* to archival storage (typically tape or other large capacity offline storage media) and reconstructs a more current state by reapplying or *redoing* the operations of committed transactions from the *backed up* log, up to the time of failure.
2. When the database on disk is not physically damaged, and a noncatastrophic failure of types 1 through 4 in Section 21.1.4 has occurred, the recovery strategy is to identify any changes that may cause an inconsistency in the database. For example, a transaction that has updated some database items on disk but has not been committed needs to have its changes reversed by *undoing* its write operations. It may also be necessary to *redo* some operations in order to restore a consistent state of the database; for example, if a transaction has committed but some of its write operations have not yet been written to disk. For noncatastrophic failure, the recovery protocol does not need a complete archival copy of the database. Rather, the entries kept in the online system log on disk are analyzed to determine the appropriate actions for recovery.

Conceptually, we can distinguish two main techniques for recovery from noncatastrophic transaction failures: deferred update and immediate update. The **deferred update** techniques do not physically update the database on disk until *after* a transaction reaches its commit point; then the updates are recorded in the database. Before reaching commit, all transaction updates are recorded in the local transaction workspace or in the main memory buffers that the DBMS maintains (the DBMS main memory cache). Before commit, the updates are recorded persistently in the log, and then after commit, the updates are written to the database on disk. 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 on disk. Hence, deferred update is also known as the **NO-UNDO/REDO algorithm**. We discuss this technique in Section 23.2.

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 must also be recorded in the log *on disk* by force-writing *before* they are applied to the database on disk, making recovery still possible. If a transaction fails after recording some changes in the database on disk 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* may be required during recovery. This technique, known as the **UNDO/REDO algorithm**, requires both operations during recovery, and is used most often in practice. A variation of the algorithm where all updates are required to be recorded in the database on disk *before* a transaction commits requires *undo* only, so it is known as the **UNDO/NO-REDO algorithm**. We discuss these techniques in Section 23.3.

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

23.1.2 Caching (Buffering) of Disk Blocks

The recovery process is often closely intertwined with operating system functions—in particular, the buffering of database disk pages in the DBMS main memory cache. Typically, multiple disk pages that include the data items to be updated are **cached** into main memory buffers 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 handled by the DBMS by calling low-level operating systems routines.

In general, it is convenient to consider recovery in terms of the database disk pages (blocks). Typically a collection of in-memory buffers, called the **DBMS cache**, is kept under the control of the DBMS for the purpose of holding these buffers. A **directory** for the cache is used to keep track of which database items are in the buffers.[1] This can be a table of <Disk_page_address, Buffer_location, ... > entries. When the DBMS requests action on some item, first it checks the cache directory to determine whether the disk page containing the item is in the DBMS cache. If it is

[1]This is somewhat similar to the concept of page tables used by the operating system.

not, the item must be located on disk, and the appropriate disk pages are copied into the cache. It may be necessary to **replace** (or **flush**) some of the cache buffers to make space available for the new item. Some page replacement strategy similar to these used in operating systems, such as least recently used (LRU) or first-in-first-out (FIFO), or a new strategy that is DBMS-specific can be used to select the buffers for replacement, such as DBMIN or Least-Likely-to-Use (see bibliographic notes).

The entries in the DBMS cache directory hold additional information relevant to buffer management. Associated with each buffer in the cache is a **dirty bit**, which can be included in the directory entry, to indicate whether or not the buffer has been modified. When a page is first read from the database disk into a cache buffer, a new entry is inserted in the cache directory with the new disk page address, and the dirty bit is set to 0 (zero). As soon as the buffer is modified, the dirty bit for the corresponding directory entry is set to 1 (one). Additional information, such as the transaction id(s) of the transaction(s) that modified the buffer can also be kept in the directory. When the buffer contents are replaced (flushed) from the cache, the contents must first be written back to the corresponding disk page *only if its dirty bit is 1.* Another bit, called the **pin-unpin** bit, is also needed—a page in the cache is **pinned** (bit value 1 (one)) if it cannot be written back to disk as yet. For example, the recovery protocol may restrict certain buffer pages from being written back to the disk until the transactions that changed this buffer have committed.

Two main strategies can be employed when flushing a modified buffer back to disk. The first strategy, known as **in-place updating**, writes the buffer to the *same original disk location,* thus overwriting the old value of any changed data items on disk.[2] Hence, a single copy of each database disk block is maintained. The second strategy, known as **shadowing**, writes an updated buffer at a different disk location, so multiple versions of data items can be maintained, but this approach is not typically used in practice.

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**). If shadowing is used, both the BFIM and the AFIM can be kept on disk; hence, it is not strictly necessary to maintain a log for recovering. We briefly discuss recovery based on shadowing in Section 23.4.

23.1.3 Write-Ahead Logging, Steal/No-Steal, and Force/No-Force

When in-place updating is used, it is necessary to use a log for recovery (see Section 21.2.2). In this case, the recovery mechanism must ensure that the BFIM of the data item is recorded in the appropriate log entry and that the log entry is flushed to disk before the BFIM is overwritten with the AFIM in the database on disk. This process is generally known as **write-ahead logging**, and is necessary to be able to UNDO the operation if this is required during recovery. Before we can describe a protocol for

[2]In-place updating is used in most systems in practice.

write-ahead logging, we need to distinguish between two types of log entry information included for a write command: the information needed for UNDO and the information needed for REDO. A **REDO-type log entry** includes the **new value** (AFIM) of the item written by the operation since this is needed to *redo* the effect of the operation from the log (by setting the item value in the database on disk to its AFIM). The **UNDO-type log entries** include the **old value** (BFIM) of the item since this is needed to *undo* the effect of the operation from the log (by setting the item value in the database back to its BFIM). In an UNDO/REDO algorithm, both types of log entries are combined. Additionally, when cascading rollback is possible, read_item entries in the log are considered to be UNDO-type entries (see Section 23.1.5).

As mentioned, the DBMS cache holds the cached database disk blocks in main memory buffers, which include not only *data blocks*, but also *index blocks* and *log blocks* from the disk. When a log record is written, it is stored in the current log buffer in the DBMS cache. The log is simply a sequential (append-only) disk file, and the DBMS cache may contain several log blocks in main memory buffers (typically, the last *n* log blocks of the log file). When an update to a data block—stored in the DBMS cache—is made, an associated log record is written to the last log buffer in the DBMS cache. With the write-ahead logging approach, the log buffers (blocks) that contain the associated log records for a particular data block update *must first be written to disk* before the data block itself can be written back to disk from its main memory buffer.

Standard DBMS recovery terminology includes the terms **steal/no-steal** and **force/no-force**, which specify the rules that govern *when* a page from the database can be written to disk from the cache:

1. If a cache buffer page updated by a transaction *cannot* be written to disk before the transaction commits, the recovery method is called a **no-steal approach**. The pin-unpin bit will be used to indicate if a page cannot be written back to disk. On the other hand, if the recovery protocol allows writing an updated buffer *before* the transaction commits, it is called **steal**. Steal is used when the DBMS cache (buffer) manager needs a buffer frame for another transaction and the buffer manager replaces an existing page that had been updated but whose transaction has not committed. The *no-steal rule* means that UNDO will never be needed during recovery, since a committed transaction will not have any of its updates on disk before it commits.
2. If all pages updated by a transaction are immediately written to disk *before* the transaction commits, it is called a **force approach**. Otherwise, it is called **no-force**. The *force rule* means that REDO will never be needed during recovery, since any committed transaction will have all its updates on disk before it is committed.

The deferred update (NO-UNDO) recovery scheme discussed in Section 23.2 follows a *no-steal* approach. However, typical database systems employ a *steal/no-force* strategy. The *advantage of steal* is that it avoids the need for a very large buffer space to store all updated pages in memory. The *advantage of no-force* is that an updated

page of a committed transaction may still be in the buffer when another transaction needs to update it, thus eliminating the I/O cost to write that page multiple times to disk, and possibly to have to read it again from disk. This may provide a substantial saving in the number of disk I/O operations when a specific page is updated heavily by multiple transactions.

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 in the database on disk until all UNDO-type log records for the updating transaction—up to this point—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 may need to maintain 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, and it may also include lists of all **committed** and **aborted transactions** since the last checkpoint (see the next section). Maintaining these lists makes the recovery process more efficient.

23.1.4 Checkpoints in the System Log and Fuzzy Checkpointing

Another type of entry in the log is called a **checkpoint**.[3] A [checkpoint, *list of active transactions*] record is written into the log periodically at that point when the system writes out to the database on disk all DBMS buffers that have been modified. As a consequence of this, 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, since all their updates will be recorded in the database on disk during checkpointing. As part of checkpointing, the list of transaction ids for active transactions at the time of the checkpoint is included in the checkpoint record, so that these transactions can be easily identified during recovery.

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 in 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 main memory buffers that have been modified to disk.

[3]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 entirely different concepts.

3. Write a [checkpoint] record to the log, and force-write the log to disk.
4. Resume executing transactions.

As a consequence of step 2, 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.

The time needed to force-write all modified memory buffers may delay transaction processing because of step 1. To reduce this delay, it is common to use a technique called **fuzzy checkpointing**. In this technique, the system can resume transaction processing after a [begin_checkpoint] record is written to the log without having to wait for step 2 to finish. When step 2 is completed, an [end_checkpoint, ...] record is written in the log with the relevant information collected during checkpointing. However, until step 2 is completed, the previous checkpoint record should remain valid. To accomplish this, the system maintains a file on disk that contains a pointer to the valid checkpoint, which continues to point to the previous checkpoint record in the log. Once step 2 is concluded, that pointer is changed to point to the new checkpoint in the log.

23.1.5 Transaction Rollback and Cascading Rollback

If a transaction fails for whatever reason after updating the database, but before the transaction commits, it may be necessary to **roll back** the transaction. If any data item values have been changed by the transaction and written to the database, they must be restored to their previous values (BFIMs). The undo-type log entries are used to restore 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 21.4.2). Understandably, cascading rollback can be quite complex and time-consuming. That is why almost all recovery mechanisms are designed so that cascading rollback *is never required.*

Figure 23.1 shows an example where cascading rollback is required. The read and write operations of three individual transactions are shown in Figure 23.1(a). Figure 23.1(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 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 23.1(b), are the T_3 operations that are undone during transaction rollback. Figure 23.1(c) graphically shows the operations of the different transactions along the time axis.

(a)

T_1
read_item(A)
read_item(D)
write_item(D)

T_2
read_item(B)
write_item(B)
read_item(D)
write_item(D)

T_3
read_item(C)
write_item(B)
read_item(A)
write_item(A)

(b)

		A	B	C	D
		30	15	40	20
	[start_transaction,T_3]				
	[read_item,T_3,C]				
*	[write_item,T_3,B,15,12]		12		
	[start_transaction,T_2]				
	[read_item,T_2,B]				
**	[write_item,T_2,B,12,18]		18		
	[start_transaction,T_1]				
	[read_item,T_1,A]				
	[read_item,T_1,D]				
	[write_item,T_1,D,20,25]				25
	[read_item,T_2,D]				
**	[write_item,T_2,D,25,26]				26
	[read_item,T_3,A]				

System crash

* T_3 is rolled back because it did not reach its commit point.

** T_2 is rolled back because it reads the value of item B written by T_3.

(c)

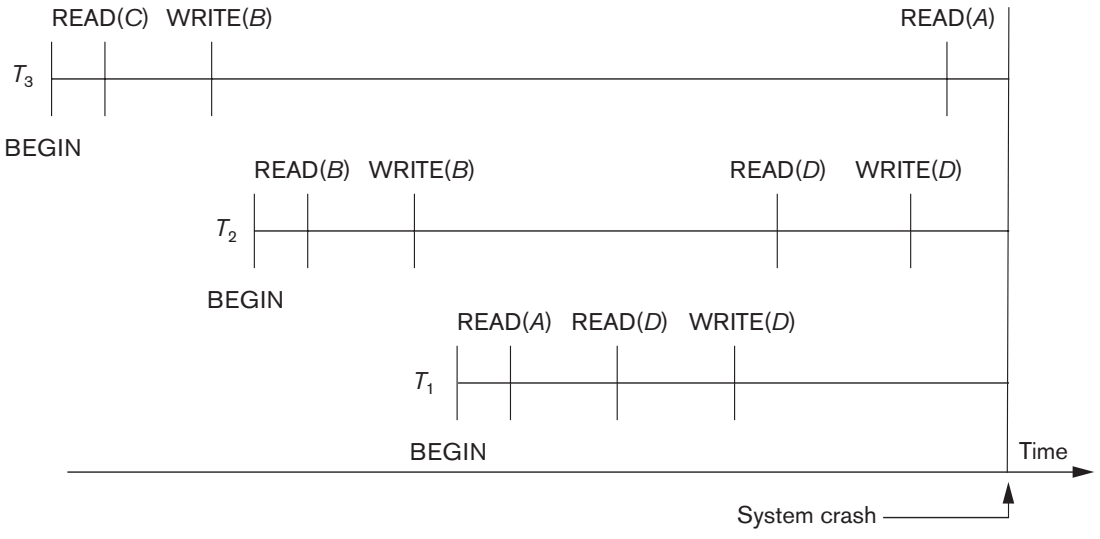


Figure 23.1
Illustrating cascading rollback (a process that never occurs in strict or cascadeless schedules). (a) The read and write operations of three transactions. (b) System log at point of crash. (c) Operations before the crash.

We must now check for cascading rollback. From Figure 23.1(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.

In practice, cascading rollback of transactions is *never* required because practical recovery methods *guarantee cascadeless or strict* schedules. Hence, there is also no need to record any read_item operations in the log because these are needed only for determining cascading rollback.

23.1.6 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. If such erroneous reports are produced, part of the recovery process would have to inform the user that these reports are wrong, since the user may take an action based on these reports that affects the database. 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, which are executed only after the transaction reaches its commit point. If the transaction fails, the batch jobs are canceled.

23.2 NO-UNDO/REDO Recovery Based on Deferred Update

The idea behind deferred update is to defer or postpone any actual updates to the database on disk until the transaction completes its execution successfully and reaches its commit point.[4]

During transaction execution, the updates are recorded only in the log and in the cache buffers. After the transaction reaches its commit point and the log is force-written to disk, the updates are recorded in the database. 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 on disk in any way. Therefore, only **REDO-type log entries** are needed in the log, which include the **new value** (AFIM) of the item written by a write operation. The **UNDO-type log entries** are not needed since no undoing of operations will be required during recovery. Although this may simplify the recovery process, it cannot be used in practice unless transactions are short

[4]Hence deferred update can generally be characterized as a *no-steal approach.*

and each transaction changes few items. For other types of transactions, there is the potential for running out of buffer space because transaction changes must be held in the cache buffers until the commit point.

We can state a typical deferred update protocol as follows:

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

Notice that step 2 of this protocol is a restatement of the write-ahead logging (WAL) protocol. Because the database is never updated on disk until after the transaction commits, there is never a need to UNDO any operations. REDO is needed in case the system fails after a transaction commits but before all its changes are recorded in the database on disk. In this case, the transaction operations are redone from the log entries during recovery.

For multiuser systems with concurrency control, the concurrency control and recovery processes are interrelated. Consider a system in which concurrency control uses strict two-phase locking, so the locks on items remain 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.

> **Procedure RDU_M (NO-UNDO/REDO with checkpoints).** Use two lists of transactions maintained by the system: the committed transactions T since the last checkpoint (**commit list**), and the active transactions T' (**active list**). 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.

The REDO procedure is defined as follows:

> **Procedure 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).

Figure 23.2 illustrates a timeline for 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. The 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 on disk under the deferred update protocol.

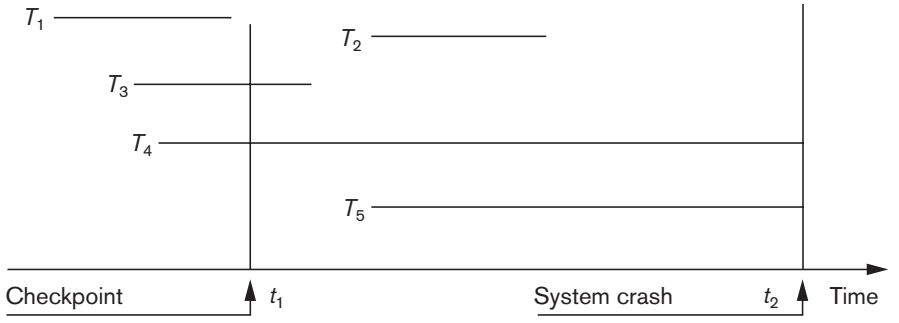


Figure 23.2
An example of a recovery timeline to illustrate the effect of checkpointing.

We can make the NO-UNDO/REDO recovery algorithm *more efficient* by noting that, if a data item *X* has been updated—as indicated in the log entries—more than once by committed transactions since the last checkpoint, it is only necessary to REDO *the last update of X* from the log during recovery because the other updates would be overwritten by this last REDO. 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 again, since its last value has already been recovered.

If a transaction is aborted for any reason (say, by the deadlock detection method), it is simply resubmitted, since it has not changed the database on disk. A drawback of the method described here is that it limits the concurrent execution of transactions because *all write-locked items remain locked until the transaction reaches its commit point.* Additionally, it may require excessive buffer space to hold all updated items until the transactions commit. The method's main benefit is that transaction operations *never need to be undone,* for two reasons:

1. A transaction does not record any changes in the database on disk until after 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 23.3 shows an example of recovery for a multiuser system that utilizes the recovery and concurrency control method just described.

23.3 Recovery Techniques Based on Immediate Update

In these techniques, when a transaction issues an update command, the database on disk can be updated *immediately*, without any need to wait for the transaction to reach its commit point. Notice that it is *not a requirement* that every update be

(a)

T_1
read_item(A)
read_item(D)
write_item(D)

T_2
read_item(B)
write_item(B)
read_item(D)
write_item(D)

T_3
read_item(A)
write_item(A)
read_item(C)
write_item(C)

T_4
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

T_2 and T_3 are ignored because they did not reach their commit points.

T_4 is redone because its commit point is after the last system checkpoint.

Figure 23.3
An example of recovery using deferred update with concurrent transactions. (a) The READ and WRITE operations of four transactions. (b) System log at the point of crash.

applied immediately to disk; it is just possible that some updates are applied to disk *before the transaction commits*.

Provisions must be made for *undoing* the effect of update operations that have been applied to the database by a *failed transaction.* This is accomplished by rolling back the transaction and undoing the effect of the transaction's write_item operations. Therefore, the **UNDO-type log entries**, which include the **old value** (BFIM) of the item, must be stored in the log. Because UNDO can be needed during recovery, these methods follow a **steal strategy** for deciding when updated main memory buffers can be written back to disk (see Section 23.1.3). Theoretically, 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. This is called the **UNDO/NO-REDO recovery algorithm**. In this method, all updates by a transaction must be recorded on disk *before the transaction commits*, so that REDO is never needed. Hence, this method must utilize the **force**

strategy for deciding when updated main memory buffers are written back to disk (see Section 23.1.3).

If the transaction is allowed to commit before all its changes are written to the database, we have the most general case, known as the **UNDO/REDO recovery algorithm**. In this case, the **steal/no-force strategy** is applied (see Section 23.1.3). This is also the most complex technique. We will outline an UNDO/REDO recovery algorithm and leave it as an exercise for the reader to develop the UNDO/NO-REDO variation. In Section 23.5, we describe a more practical approach known as the ARIES recovery technique.

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 algorithm for concurrent transactions with immediate update (UNDO/REDO recovery). 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 (or aborted and rolled back). However, deadlocks can occur in strict two-phase locking, thus requiring abort and UNDO of transactions. For a strict schedule, UNDO of an operation requires changing the item back to its old value (BFIM).

Procedure RIU_M (UNDO/REDO with checkpoints).

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, using the REDO procedure defined earlier.

The UNDO procedure is defined as follows:

Procedure 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.

As we discussed for the **NO-UNDO/REDO** procedure, step 3 is more efficiently done by starting from the *end of the log* and redoing only *the last update of each item X*. Whenever an item is redone, it is added to a list of redone items and is not redone again. A similar procedure can be devised to improve the efficiency of step 2 so that an item can be undone at most once during recovery. In this case, the earliest UNDO is applied first by scanning the log in the forward direction (starting from the

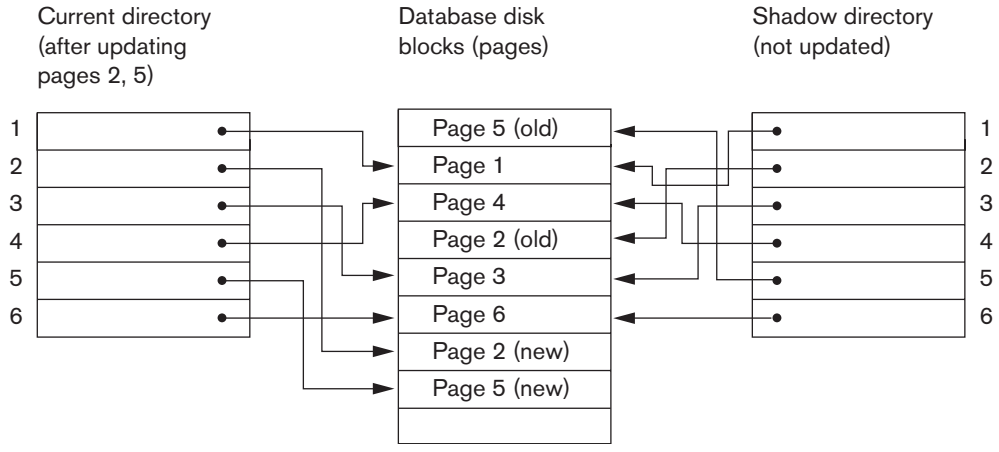
beginning of the log). Whenever an item is undone, it is added to a list of undone items and is not undone again.

23.4 Shadow Paging

This recovery scheme does not require the use of a log in a single-user environment. In a multiuser environment, a log may be 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 **directory** with *n* entries[5] is constructed, where the *i*th entry points to the *i*th database page on disk. The directory is kept in main memory if it is not too large, and all references—reads or writes—to database pages on disk go through it. When a transaction begins executing, the **current directory**—whose entries point to the most recent or current database pages on disk—is copied into a **shadow directory**. The shadow directory is then saved on disk while the current directory is used by the transaction.

During transaction execution, the shadow directory 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 directory entry is modified to point to the new disk block, whereas the shadow directory is not modified and continues to point to the old unmodified disk block. Figure 23.4 illustrates the concepts of shadow and current directories. For pages updated by the transaction, two versions are kept. The old version is referenced by the shadow directory and the new version by the current directory.

Figure 23.4
An example of shadow paging.



[5]The directory is similar to the page table maintained by the operating system for each process.

To recover from a failure during transaction execution, it is sufficient to free the modified database pages and to discard the current directory. The state of the database before transaction execution is available through the shadow directory, and that state is recovered by reinstating the shadow directory. 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 directory. Since recovery involves neither undoing nor redoing data items, this technique can be categorized as a NO-UNDO/NO-REDO technique for recovery.

In a multiuser environment with concurrent transactions, logs and checkpoints must be incorporated into the shadow paging technique. One disadvantage of shadow paging is 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 directory is large, the overhead of writing shadow directories 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 directory 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. Another issue is that the operation to migrate between current and shadow directories must be implemented as an atomic operation.

23.5 The ARIES Recovery Algorithm

We now describe the ARIES algorithm as an example of a recovery algorithm used in database systems. It is used in many relational database-related products of IBM. ARIES uses a steal/no-force approach for writing, and it is based on three concepts: write-ahead logging, repeating history during redo, and logging changes during undo. We discussed write-ahead logging in Section 23.1.3. The second concept, **repeating history**, means that ARIES will retrace all actions of the database system prior to the crash to reconstruct the database state *when the crash occurred*. Transactions that were uncommitted at the time of the crash (active transactions) are undone. The third concept, **logging during undo**, will prevent ARIES from repeating the completed undo operations if a failure occurs during recovery, which causes a restart of the recovery process.

The ARIES recovery procedure consists of three main steps: analysis, REDO, and UNDO. The **analysis step** identifies the dirty (updated) pages in the buffer[6] and the set of transactions active at the time of the crash. The appropriate point in the log where the REDO operation should start is also determined. The **REDO phase** actually reapplies updates from the log to the database. Generally, the REDO operation is applied only to committed transactions. However, this is not the case in ARIES. Certain information in the ARIES log will provide the start point for REDO, from

[6]The actual buffers may be lost during a crash, since they are in main memory. Additional tables stored in the log during checkpointing (Dirty Page Table, Transaction Table) allows ARIES to identify this information (as discussed later in this section).

which REDO operations are applied until the end of the log is reached. Additionally, information stored by ARIES and in the data pages will allow ARIES to determine whether the operation to be redone has actually been applied to the database and therefore does not need to be reapplied. Thus, *only the necessary REDO operations* are applied during recovery. Finally, during the **UNDO phase**, the log is scanned backward and the operations of transactions that were active at the time of the crash are undone in reverse order. The information needed for ARIES to accomplish its recovery procedure includes the log, the Transaction Table, and the Dirty Page Table. Additionally, checkpointing is used. These tables are maintained by the transaction manager and written to the log during checkpointing.

In ARIES, every log record has an associated **log sequence number** (**LSN**) that is monotonically increasing and indicates the address of the log record on disk. Each LSN corresponds to a *specific change* (action) of some transaction. Also, each data page will store the LSN of the *latest log record corresponding to a change for that page.* A log record is written for any of the following actions: updating a page (write), committing a transaction (commit), aborting a transaction (abort), undoing an update (undo), and ending a transaction (end). The need for including the first three actions in the log has been discussed, but the last two need some explanation. When an update is undone, a *compensation log record* is written in the log. When a transaction ends, whether by committing or aborting, an *end log record* is written.

Common fields in all log records include the previous LSN for that transaction, the transaction ID, and the type of log record. The previous LSN is important because it links the log records (in reverse order) for each transaction. For an update (write) action, additional fields in the log record include the page ID for the page that contains the item, the length of the updated item, its offset from the beginning of the page, the before image of the item, and its after image.

Besides the log, two tables are needed for efficient recovery: the **Transaction Table** and the **Dirty Page Table**, which are maintained by the transaction manager. When a crash occurs, these tables are rebuilt in the analysis phase of recovery. The Transaction Table contains an entry for *each active transaction,* with information such as the transaction ID, transaction status, and the LSN of the most recent log record for the transaction. The Dirty Page Table contains an entry for each dirty page in the buffer, which includes the page ID and the LSN corresponding to the earliest update to that page.

Checkpointing in ARIES consists of the following: writing a begin_checkpoint record to the log, writing an end_checkpoint record to the log, and writing *the LSN of* the begin_checkpoint record to a special file. This special file is accessed during recovery to locate the last checkpoint information. With the end_checkpoint record, the contents of both the Transaction Table and Dirty Page Table are appended to the end of the log. To reduce the cost, **fuzzy checkpointing** is used so that the DBMS can continue to execute transactions during checkpointing (see Section 23.1.4). Additionally, the contents of the DBMS cache do not have to be flushed to disk during checkpoint, since the Transaction Table and Dirty Page Table—which are appended to the log on disk—contain the information needed for recovery. Note

that if a crash occurs during checkpointing, the special file will refer to the previous checkpoint, which is used for recovery.

After a crash, the ARIES recovery manager takes over. Information from the last checkpoint is first accessed through the special file. The **analysis phase** starts at the begin_checkpoint record and proceeds to the end of the log. When the end_checkpoint record is encountered, the Transaction Table and Dirty Page Table are accessed (recall that these tables were written in the log during checkpointing). During analysis, the log records being analyzed may cause modifications to these two tables. For instance, if an end log record was encountered for a transaction T in the Transaction Table, then the entry for T is deleted from that table. If some other type of log record is encountered for a transaction T', then an entry for T' is inserted into the Transaction Table, if not already present, and the last LSN field is modified. If the log record corresponds to a change for page P, then an entry would be made for page P (if not present in the table) and the associated LSN field would be modified. When the analysis phase is complete, the necessary information for REDO and UNDO has been compiled in the tables.

The **REDO phase** follows next. To reduce the amount of unnecessary work, ARIES starts redoing at a point in the log where it knows (for sure) that previous changes to dirty pages *have already been applied to the database on disk.* It can determine this by finding the smallest LSN, M, of all the dirty pages in the Dirty Page Table, which indicates the log position where ARIES needs to start the REDO phase. Any changes corresponding to an LSN $< M$, for redoable transactions, must have already been propagated to disk or already been overwritten in the buffer; otherwise, those dirty pages with that LSN would be in the buffer (and the Dirty Page Table). So, REDO starts at the log record with LSN = M and scans forward to the end of the log. For each change recorded in the log, the REDO algorithm would verify whether or not the change has to be reapplied. For example, if a change recorded in the log pertains to page P that is not in the Dirty Page Table, then this change is already on disk and does not need to be reapplied. Or, if a change recorded in the log (with LSN = N, say) pertains to page P and the Dirty Page Table contains an entry for P with LSN greater than N, then the change is already present. If neither of these two conditions hold, page P is read from disk and the LSN stored on that page, LSN(P), is compared with N. If $N <$ LSN(P), then the change has been applied and the page does not need to be rewritten to disk.

Once the REDO phase is finished, the database is in the exact state that it was in when the crash occurred. The set of active transactions—called the undo_set—has been identified in the Transaction Table during the analysis phase. Now, the **UNDO phase** proceeds by scanning backward from the end of the log and undoing the appropriate actions. A compensating log record is written for each action that is undone. The UNDO reads backward in the log until every action of the set of transactions in the undo_set has been undone. When this is completed, the recovery process is finished and normal processing can begin again.

Consider the recovery example shown in Figure 23.5. There are three transactions: T_1, T_2, and T_3. T_1 updates page C, T_2 updates pages B and C, and T_3 updates page A.

(a)

Lsn	Last_lsn	Tran_id	Type	Page_id	Other_information
1	0	T_1	update	*C*	...
2	0	T_2	update	*B*	...
3	1	T_1	commit		...
4	begin checkpoint				
5	end checkpoint				
6	0	T_3	update	*A*	...
7	2	T_2	update	*C*	...
8	7	T_2	commit		...

(b)

TRANSACTION TABLE

Transaction_id	Last_lsn	Status
T_1	3	commit
T_2	2	in progress

DIRTY PAGE TABLE

Page_id	Lsn
C	1
B	2

(c)

TRANSACTION TABLE

Transaction_id	Last_lsn	Status
T_1	3	commit
T_2	8	commit
T_3	6	in progress

DIRTY PAGE TABLE

Page_id	Lsn
C	1
B	2
A	6

Figure 23.5
An example of recovery in ARIES. (a) The log at point of crash. (b) The Transaction and Dirty Page Tables at time of checkpoint. (c) The Transaction and Dirty Page Tables after the analysis phase.

Figure 23.5(a) shows the partial contents of the log, and Figure 23.5(b) shows the contents of the Transaction Table and Dirty Page Table. Now, suppose that a crash occurs at this point. Since a checkpoint has occurred, the address of the associated begin_checkpoint record is retrieved, which is location 4. The analysis phase starts from location 4 until it reaches the end. The end_checkpoint record would contain the Transaction Table and Dirty Page Table in Figure 23.5(b), and the analysis phase will further reconstruct these tables. When the analysis phase encounters log record 6, a new entry for transaction T_3 is made in the Transaction Table and a new entry for page A is made in the Dirty Page Table. After log record 8 is analyzed, the status of transaction T_2 is changed to committed in the Transaction Table. Figure 23.5(c) shows the two tables after the analysis phase.

For the REDO phase, the smallest LSN in the Dirty Page Table is 1. Hence the REDO will start at log record 1 and proceed with the REDO of updates. The LSNs {1, 2, 6, 7} corresponding to the updates for pages C, B, A, and C, respectively, are not less than the LSNs of those pages (as shown in the Dirty Page Table). So those data pages will be read again and the updates reapplied from the log (assuming the actual LSNs stored on those data pages are less then the corresponding log entry). At this point, the REDO phase is finished and the UNDO phase starts. From the Transaction Table (Figure 23.5(c)), UNDO is applied only to the active transaction T_3. The UNDO phase starts at log entry 6 (the last update for T_3) and proceeds backward in the log. The backward chain of updates for transaction T_3 (only log record 6 in this example) is followed and undone.

23.6 Recovery in Multidatabase Systems

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 object-oriented, hierarchical, or network DBMSs. In such a case, each DBMS involved in the multidatabase transaction may 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 25), 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 to maintain information needed for recovery, in addition to the local recovery managers and the information they maintain (log, tables). 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 and needed information for local recovery 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 the database within a certain time out interval, it assumes a *not OK* response.
- **Phase 2.** If *all* participating databases reply *OK*, and the coordinator's vote is also *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 and information needed for local recovery 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 or the coordinator have a *not OK* response, 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.

23.7 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 directory 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 directory to recover from 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 a **database backup**, in which the whole database and the log are periodically copied onto a cheap storage medium such as magnetic tapes or other large capacity offline storage devices. 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.

Data from critical applications such as banking, insurance, stock market, and other databases is periodically backed up in its entirety and moved to physically separate safe locations. Subterranean storage vaults have been used to protect such data from flood, storm, earthquake, or fire damage. Events like the 9/11 terrorist attack in New York (in 2001) and the Katrina hurricane disaster in New Orleans (in 2005) have created a greater awareness of *disaster recovery of business-critical databases*.

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 at more frequent intervals than full database backup 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. Therefore, 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 to tape can have their effect on the database redone. A new log is started after each database backup. Hence, to recover from disk failure, the database is first recreated on disk from its latest backup copy on tape. Following that, the effects of all the committed transactions whose operations have been recorded in the backed-up copies of the system log are reconstructed.

23.8 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. First we gave an informal outline for a recovery process and then we 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, steal/no-steal and force/no-force policies, system checkpointing, 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 on disk 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. The disadvantage is that too much buffer space may be needed, since updates are kept in the buffers and are not applied to disk until a transaction commits. Deferred update can lead to a recovery algorithm known as NO-UNDO/REDO. Immediate update techniques may apply changes to the database on disk before the transaction reaches a successful conclusion. 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 directory. 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 also presented ARIES, a specific recovery scheme used in many of IBM's relational database products. Then we 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

23.1. Discuss the different types of transaction failures. What is meant by catastrophic failure?

23.2. Discuss the actions taken by the read_item and write_item operations on a database.

23.3. 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?

23.4. How are buffering and caching techniques used by the recovery subsystem?

23.5. What are the before image (BFIM) and 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?

23.6. What are UNDO-type and REDO-type log entries?

23.7. Describe the write-ahead logging protocol.

23.8. Identify three typical lists of transactions that are maintained by the recovery subsystem.

23.9. What is meant by transaction rollback? What is meant by cascading rollback? Why do practical recovery methods use protocols that do not permit cascading rollback? Which recovery techniques do not require any rollback?

23.10. Discuss the UNDO and REDO operations and the recovery techniques that use each.

23.11. Discuss the deferred update technique of recovery. What are the advantages and disadvantages of this technique? Why is it called the NO-UNDO/REDO method?

23.12. How can recovery handle transaction operations that do not affect the database, such as the printing of reports by a transaction?

23.13. Discuss the immediate update recovery technique in both single-user and multiuser environments. What are the advantages and disadvantages of immediate update?

23.14. 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.

23.15. Describe the shadow paging recovery technique. Under what circumstances does it not require a log?

23.16. Describe the three phases of the ARIES recovery method.

23.17. What are log sequence numbers (LSNs) in ARIES? How are they used? What information do the Dirty Page Table and Transaction Table contain? Describe how fuzzy checkpointing is used in ARIES.

23.18. What do the terms steal/no-steal and force/no-force mean with regard to buffer management for transaction processing?

23.19. Describe the two-phase commit protocol for multidatabase transactions.

23.20. Discuss how disaster recovery from catastrophic failures is handled.

Exercises

23.21. Suppose that the system crashes before the [read_item, T_3, A] entry is written to the log in Figure 23.1(b). Will that make any difference in the recovery process?

23.22. Suppose that the system crashes before the [write_item, T_2, D, 25, 26] entry is written to the log in Figure 23.1(b). Will that make any difference in the recovery process?

23.23. Figure 23.6 shows the log corresponding to a particular schedule at the point of a system crash for four transactions T_1, T_2, T_3, and T_4. Suppose 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.

[start_transaction, T_1]
[read_item, T_1, A]
[read_item, T_1, D]
[write_item, T_1, D, 20, 25]
[commit, T_1]
[checkpoint]
[start_transaction, T_2]
[read_item, T_2, B]
[write_item, T_2, B, 12, 18]
[start_transaction, T_4]
[read_item, T_4, D]
[write_item, T_4, D, 25, 15]
[start_transaction, T_3]
[write_item, T_3, C, 30, 40]
[read_item, T_4, A]
[write_item, T_4, A, 30, 20]
[commit, T_4]
[read_item, T_2, D]
[write_item, T_2, D, 15, 25] ← System crash

Figure 23.6
A sample schedule and its corresponding log.

23.24. Suppose that we use the deferred update protocol for the example in Figure 23.6. 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.

23.25. How does checkpointing in ARIES differ from checkpointing as described in Section 23.1.4?

23.26. How are log sequence numbers used by ARIES to reduce the amount of REDO work needed for recovery? Illustrate with an example using the information shown in Figure 23.5. You can make your own assumptions as to when a page is written to disk.

23.27. What implications would a no-steal/force buffer management policy have on checkpointing and recovery?

Choose the correct answer for each of the following multiple-choice questions:

23.28. Incremental logging with deferred updates implies that the recovery system must necessarily

a. store the old value of the updated item in the log.

b. store the new value of the updated item in the log.

c. store both the old and new value of the updated item in the log.

d. store only the Begin Transaction and Commit Transaction records in the log.

23.29. The write-ahead logging (WAL) protocol simply means that

a. writing of a data item should be done ahead of any logging operation.

b. the log record for an operation should be written before the actual data is written.

c. all log records should be written before a new transaction begins execution.

d. the log never needs to be written to disk.

23.30. In case of transaction failure under a deferred update incremental logging scheme, which of the following will be needed?

a. an undo operation

b. a redo operation

c. an undo and redo operation

d. none of the above

23.31. For incremental logging with immediate updates, a log record for a transaction would contain

a. a transaction name, a data item name, and the old and new value of the item.

b. a transaction name, a data item name, and the old value of the item.
c. a transaction name, a data item name, and the new value of the item.
d. a transaction name and a data item name.

23.32. For correct behavior during recovery, undo and redo operations must be
a. commutative.
b. associative.
c. idempotent.
d. distributive.

23.33. When a failure occurs, the log is consulted and each operation is either undone or redone. This is a problem because
a. searching the entire log is time consuming.
b. many redos are unnecessary.
c. both (a) and (b).
d. none of the above.

23.34. When using a log-based recovery scheme, it might improve performance as well as providing a recovery mechanism by
a. writing the log records to disk when each transaction commits.
b. writing the appropriate log records to disk during the transaction's execution.
c. waiting to write the log records until multiple transactions commit and writing them as a batch.
d. never writing the log records to disk.

23.35. There is a possibility of a cascading rollback when
a. a transaction writes items that have been written only by a committed transaction.
b. a transaction writes an item that is previously written by an uncommitted transaction.
c. a transaction reads an item that is previously written by an uncommitted transaction.
d. both (b) and (c).

23.36. To cope with media (disk) failures, it is necessary
a. for the DBMS to only execute transactions in a single user environment.
b. to keep a redundant copy of the database.
c. to never abort a transaction.
d. all of the above.

23.37. If the shadowing approach is used for flushing a data item back to disk, then
   a. the item is written to disk only after the transaction commits.
   b. the item is written to a different location on disk.
   c. the item is written to disk before the transaction commits.
   d. the item is written to the same disk location from which it was read.

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) discuss the recovery mechanism in SYSTEM R. Lockemann and Knutsen (1968), Davies (1973), and Bjork (1973) are early papers that discuss recovery. Chandy et al. (1975) discuss transaction rollback. Lilien and Bhargava (1985) discuss 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. 1992). 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. Two books by Kumar and Hsu (1998) and Kumar and Song (1998) discuss recovery in detail and contain descriptions of recovery methods used in a number of existing relational database products. Examples of page replacement strategies that are specific for databases are discussed in Chou and DeWitt (1985) and Pazos et al. (2006).

part 10

Additional Database Topics: Security and Distribution

chapter 24

Database Security

This chapter discusses techniques for securing databases against a variety of threats. It also presents schemes of providing access privileges to authorized users. Some of the security threats to databases—such as SQL Injection—will be presented. At the end of the chapter we also summarize how a commercial RDBMS—specifically, the Oracle system—provides different types of security. We start in Section 24.1 with an introduction to security issues and the threats to databases, and we give an overview of the control measures that are covered in the rest of this chapter. We also comment on the relationship between data security and privacy as it applies to personal information. Section 24.2 discusses the mechanisms used to grant and revoke privileges in relational database systems and in SQL, mechanisms that are often referred to as **discretionary access control**. In Section 24.3, we present an overview of the mechanisms for enforcing multiple levels of security—a particular concern in database system security that is known as **mandatory access control**. Section 24.3 also introduces the more recently developed strategies of **role-based access control**, and label-based and row-based security. Section 24.3 also provides a brief discussion of XML access control. Section 24.4 discusses a major threat to databases called SQL Injection, and discusses some of the proposed preventive measures against it. Section 24.5 briefly discusses the security problem in statistical databases. Section 24.6 introduces the topic of flow control and mentions problems associated with covert channels. Section 24.7 provides a brief summary of encryption and symmetric key and asymmetric (public) key infrastructure schemes. It also discusses digital certificates. Section 24.8 introduces privacy-preserving techniques, and Section 24.9 presents the current challenges to database security. In Section 24.10, we discuss Oracle label-based security. Finally, Section 24.11 summarizes the chapter. Readers who are interested only in basic database security mechanisms will find it sufficient to cover the material in Sections 24.1 and 24.2.

24.1 Introduction to Database Security Issues[1]

24.1.1 Types of Security

Database security is a broad area that addresses many issues, including the following:

- Various legal and ethical issues regarding the right to access certain information—for example, some information may be deemed to be private and cannot be accessed legally by unauthorized organizations or persons. In the United States, there are numerous laws governing privacy of information.
- Policy issues at the governmental, institutional, or corporate level as to what kinds of information should not be made publicly available—for example, credit ratings and personal medical records.
- System-related issues such as the *system levels* at which various security functions should be enforced—for example, whether a security function should be handled at the physical hardware level, the operating system level, or the DBMS level.
- The need in some organizations to identify multiple *security levels* and to categorize the data and users based on these classifications—for example, top secret, secret, confidential, and unclassified. The security policy of the organization with respect to permitting access to various classifications of data must be enforced.

Threats to Databases. Threats to databases can result in the loss or degradation of some or all of the following commonly accepted security goals: integrity, availability, and confidentiality.

- **Loss of integrity.** Database integrity refers to the requirement that information be protected from improper modification. Modification of data includes creation, insertion, updating, changing the status of data, and deletion. Integrity is lost if unauthorized changes are made to the data by either intentional or accidental acts. If the loss of system or data integrity is not corrected, continued use of the contaminated system or corrupted data could result in inaccuracy, fraud, or erroneous decisions.
- **Loss of availability.** Database availability refers to making objects available to a human user or a program to which they have a legitimate right.
- **Loss of confidentiality.** Database confidentiality refers to the protection of data from unauthorized disclosure. The impact of unauthorized disclosure of confidential information can range from violation of the Data Privacy Act to the jeopardization of national security. Unauthorized, unanticipated, or unintentional disclosure could result in loss of public confidence, embarrassment, or legal action against the organization.

[1]The substantial contribution of Fariborz Farahmand and Bharath Rengarajan to this and subsequent sections in this chapter is much appreciated.

To protect databases against these types of threats, it is common to implement *four kinds of control measures*: access control, inference control, flow control, and encryption. We discuss each of these in this chapter.

In a multiuser database system, the DBMS must provide techniques to enable certain users or user groups to access selected portions of a database without gaining access to the rest of the database. This is particularly important when a large integrated database is to be used by many different users within the same organization. For example, sensitive information such as employee salaries or performance reviews should be kept confidential from most of the database system's users. A DBMS typically includes a **database security and authorization subsystem** that is responsible for ensuring the security of portions of a database against unauthorized access. It is now customary to refer to two types of database security mechanisms:

- **Discretionary security mechanisms.** These are used to grant privileges to users, including the capability to access specific data files, records, or fields in a specified mode (such as read, insert, delete, or update).
- **Mandatory security mechanisms.** These are used to enforce multilevel security by classifying the data and users into various security classes (or levels) and then implementing the appropriate security policy of the organization. For example, a typical security policy is to permit users at a certain classification (or clearance) level to see only the data items classified at the user's own (or lower) classification level. An extension of this is *role-based security,* which enforces policies and privileges based on the concept of organizational roles.

We discuss discretionary security in Section 24.2 and mandatory and role-based security in Section 24.3.

24.1.2 Control Measures

Four main control measures are used to provide security of data in databases:

- Access control
- Inference control
- Flow control
- Data encryption

A security problem common to computer systems is that of preventing unauthorized persons from accessing the system itself, either to obtain information or to make malicious changes in a portion of the database. The security mechanism of a DBMS must include provisions for restricting access to the database system as a whole. This function, called **access control**, is handled by creating user accounts and passwords to control the login process by the DBMS. We discuss access control techniques in Section 24.1.3.

S**tatistical databases** are used to provide statistical information or summaries of values based on various criteria. For example, a database for population statistics

may provide statistics based on age groups, income levels, household size, education levels, and other criteria. Statistical database users such as government statisticians or market research firms are allowed to access the database to retrieve statistical information about a population but not to access the detailed confidential information about specific individuals. Security for statistical databases must ensure that information about individuals cannot be accessed. It is sometimes possible to deduce or infer certain facts concerning individuals from queries that involve only summary statistics on groups; consequently, this must not be permitted either. This problem, called **statistical database security**, is discussed briefly in Section 24.4. The corresponding control measures are called **inference control** measures.

Another security issue is that of **flow control**, which prevents information from flowing in such a way that it reaches unauthorized users. It is discussed in Section 24.6. Channels that are pathways for information to flow implicitly in ways that violate the security policy of an organization are called **covert channels**. We briefly discuss some issues related to covert channels in Section 24.6.1.

A final control measure is **data encryption**, which is used to protect sensitive data (such as credit card numbers) that is transmitted via some type of communications network. Encryption can be used to provide additional protection for sensitive portions of a database as well. The data is **encoded** using some coding algorithm. An unauthorized user who accesses encoded data will have difficulty deciphering it, but authorized users are given decoding or decrypting algorithms (or keys) to decipher the data. Encrypting techniques that are very difficult to decode without a key have been developed for military applications. Section 24.7 briefly discusses encryption techniques, including popular techniques such as public key encryption, which is heavily used to support Web-based transactions against databases, and digital signatures, which are used in personal communications.

A comprehensive discussion of security in computer systems and databases is outside the scope of this textbook. We give only a brief overview of database security techniques here. The interested reader can refer to several of the references discussed in the Selected Bibliography at the end of this chapter for a more comprehensive discussion.

24.1.3 Database Security and the DBA

As we discussed in Chapter 1, the database administrator (DBA) is the central authority for managing a database system. The DBA's responsibilities include granting privileges to users who need to use the system and classifying users and data in accordance with the policy of the organization. The DBA has a **DBA account** in the DBMS, sometimes called a **system** or **superuser account**, which provides powerful capabilities that are not made available to regular database accounts and users.[2] DBA-privileged commands include commands for granting and revoking privileges

[2]This account is similar to the *root* or *superuser* accounts that are given to computer system administrators, which allow access to restricted operating system commands.

to individual accounts, users, or user groups and for performing the following types of actions:

1. **Account creation.** This action creates a new account and password for a user or a group of users to enable access to the DBMS.
2. **Privilege granting.** This action permits the DBA to grant certain privileges to certain accounts.
3. **Privilege revocation.** This action permits the DBA to revoke (cancel) certain privileges that were previously given to certain accounts.
4. **Security level assignment.** This action consists of assigning user accounts to the appropriate security clearance level.

The DBA is responsible for the overall security of the database system. Action 1 in the preceding list is used to control access to the DBMS as a whole, whereas actions 2 and 3 are used to control *discretionary* database authorization, and action 4 is used to control *mandatory* authorization.

24.1.4 Access Control, User Accounts, and Database Audits

Whenever a person or a group of persons needs to access a database system, the individual or group must first apply for a user account. The DBA will then create a new **account number** and **password** for the user if there is a legitimate need to access the database. The user must **log in** to the DBMS by entering the account number and password whenever database access is needed. The DBMS checks that the account number and password are valid; if they are, the user is permitted to use the DBMS and to access the database. Application programs can also be considered users and are required to log in to the database (see Chapter 13).

It is straightforward to keep track of database users and their accounts and passwords by creating an encrypted table or file with two fields: AccountNumber and Password. This table can easily be maintained by the DBMS. Whenever a new account is created, a new record is inserted into the table. When an account is canceled, the corresponding record must be deleted from the table.

The database system must also keep track of all operations on the database that are applied by a certain user throughout each **login session**, which consists of the sequence of database interactions that a user performs from the time of logging in to the time of logging off. When a user logs in, the DBMS can record the user's account number and associate it with the computer or device from which the user logged in. All operations applied from that computer or device are attributed to the user's account until the user logs off. It is particularly important to keep track of update operations that are applied to the database so that, if the database is tampered with, the DBA can determine which user did the tampering.

To keep a record of all updates applied to the database and of particular users who applied each update, we can modify the *system log.* Recall from Chapters 21 and 23 that the **system log** includes an entry for each operation applied to the database that may be required for recovery from a transaction failure or system crash. We can

expand the log entries so that they also include the account number of the user and the online computer or device ID that applied each operation recorded in the log. If any tampering with the database is suspected, a **database audit** is performed, which consists of reviewing the log to examine all accesses and operations applied to the database during a certain time period. When an illegal or unauthorized operation is found, the DBA can determine the account number used to perform the operation. Database audits are particularly important for sensitive databases that are updated by many transactions and users, such as a banking database that is updated by many bank tellers. A database log that is used mainly for security purposes is sometimes called an **audit trail**.

24.1.5 Sensitive Data and Types of Disclosures

Sensitivity of data is a measure of the importance assigned to the data by its owner, for the purpose of denoting its need for protection. Some databases contain only sensitive data while other databases may contain no sensitive data at all. Handling databases that fall at these two extremes is relatively easy, because these can be covered by access control, which is explained in the next section. The situation becomes tricky when some of the data is sensitive while other data is not.

Several factors can cause data to be classified as sensitive:

1. **Inherently sensitive.** The value of the data itself may be so revealing or confidential that it becomes sensitive—for example, a person's salary or that a patient has HIV/AIDS.
2. **From a sensitive source.** The source of the data may indicate a need for secrecy—for example, an informer whose identity must be kept secret.
3. **Declared sensitive.** The owner of the data may have explicitly declared it as sensitive.
4. **A sensitive attribute or sensitive record.** The particular attribute or record may have been declared sensitive—for example, the salary attribute of an employee or the salary history record in a personnel database.
5. **Sensitive in relation to previously disclosed data.** Some data may not be sensitive by itself but will become sensitive in the presence of some other data—for example, the exact latitude and longitude information for a location where some previously recorded event happened that was later deemed sensitive.

It is the responsibility of the database administrator and security administrator to collectively enforce the security policies of an organization. This dictates whether access should be permitted to a certain database attribute (also known as a *table column* or a *data element*) or not for individual users or for categories of users. Several factors need to be considered before deciding whether it is safe to reveal the data. The three most important factors are data availability, access acceptability, and authenticity assurance.

1. **Data availability.** If a user is updating a field, then this field becomes inaccessible and other users should not be able to view this data. This blocking is

only temporary and only to ensure that no user sees any inaccurate data. This is typically handled by the concurrency control mechanism (see Chapter 22).

2. **Access acceptability.** Data should only be revealed to authorized users. A database administrator may also deny access to a user request even if the request does not directly access a sensitive data item, on the grounds that the requested data may reveal information about the sensitive data that the user is not authorized to have.
3. **Authenticity assurance.** Before granting access, certain external characteristics about the user may also be considered. For example, a user may only be permitted access during working hours. The system may track previous queries to ensure that a combination of queries does not reveal sensitive data. The latter is particularly relevant to statistical database queries (see Section 24.5).

The term *precision*, when used in the security area, refers to allowing as much as possible of the data to be available, subject to protecting exactly the subset of data that is sensitive. The definitions of *security* versus *precision* are as follows:

- **Security:** Means of ensuring that data is kept safe from corruption and that access to it is suitably controlled. To provide security means to disclose only nonsensitive data, and reject any query that references a sensitive field.
- **Precision**: To protect all sensitive data while disclosing as much nonsensitive data as possible.

The ideal combination is to maintain perfect security with maximum precision. If we want to maintain security, some sacrifice has to be made with precision. Hence there is typically a tradeoff between security and precision.

24.1.6 Relationship between Information Security versus Information Privacy

The rapid advancement of the use of information technology (IT) in industry, government, and academia raises challenging questions and problems regarding the protection and use of personal information. Questions of *who* has *what* rights to information about individuals for *which* purposes become more important as we move toward a world in which it is technically possible to know just about anything about anyone.

Deciding how to design privacy considerations in technology for the future includes philosophical, legal, and practical dimensions. There is a considerable overlap between issues related to access to resources (security) and issues related to appropriate use of information (privacy). We now define the difference between *security* versus *privacy*.

Security in information technology refers to many aspects of protecting a system from unauthorized use, including authentication of users, information encryption, access control, firewall policies, and intrusion detection. For our purposes here, we

will limit our treatment of security to the concepts associated with how well a system can protect access to information it contains. The concept of **privacy** goes beyond security. Privacy examines how well the use of personal information that the system acquires about a user conforms to the explicit or implicit assumptions regarding that use. From an end user perspective, privacy can be considered from two different perspectives: *preventing storage* of personal information versus *ensuring appropriate use* of personal information.

For the purposes of this chapter, a simple but useful definition of **privacy** is *the ability of individuals to control the terms under which their personal information is acquired and used*. In summary, security involves technology to ensure that information is appropriately protected. Security is a required building block for privacy to exist. Privacy involves mechanisms to support compliance with some basic principles and other explicitly stated policies. One basic principle is that people should be informed about information collection, told in advance what will be done with their information, and given a reasonable opportunity to approve of such use of the information. A related concept**, trust**, relates to both security and privacy, and is seen as increasing when it is perceived that both security and privacy are provided for.

24.2 Discretionary Access Control Based on Granting and Revoking Privileges

The typical method of enforcing **discretionary access control** in a database system is based on the granting and revoking of **privileges**. Let us consider privileges in the context of a relational DBMS. In particular, we will discuss a system of privileges somewhat similar to the one originally developed for the SQL language (see Chapters 4 and 5). Many current relational DBMSs use some variation of this technique. The main idea is to include statements in the query language that allow the DBA and selected users to grant and revoke privileges.

24.2.1 Types of Discretionary Privileges

In SQL2 and later versions,[3] the concept of an **authorization identifier** is used to refer, roughly speaking, to a user account (or group of user accounts). For simplicity, we will use the words *user* or *account* interchangeably in place of *authorization identifier*. The DBMS must provide selective access to each relation in the database based on specific accounts. Operations may also be controlled; thus, having an account does not necessarily entitle the account holder to all the functionality provided by the DBMS. Informally, there are two levels for assigning privileges to use the database system:

- **The account level.** At this level, the DBA specifies the particular privileges that each account holds independently of the relations in the database.
- **The relation (or table) level.** At this level, the DBA can control the privilege to access each individual relation or view in the database.

[3]Discretionary privileges were incorporated into SQL2 and are applicable to later versions of SQL.

The privileges at the **account level** apply to the capabilities provided to the account itself and can include the CREATE SCHEMA or CREATE TABLE privilege, to create a schema or base relation; the CREATE VIEW privilege; the ALTER privilege, to apply schema changes such as adding or removing attributes from relations; the DROP privilege, to delete relations or views; the MODIFY privilege, to insert, delete, or update tuples; and the SELECT privilege, to retrieve information from the database by using a SELECT query. Notice that these account privileges apply to the account in general. If a certain account does not have the CREATE TABLE privilege, no relations can be created from that account. Account-level privileges *are not* defined as part of SQL2; they are left to the DBMS implementers to define. In earlier versions of SQL, a CREATETAB privilege existed to give an account the privilege to create tables (relations).

The second level of privileges applies to the **relation level**, whether they are base relations or virtual (view) relations. These privileges *are* defined for SQL2. In the following discussion, the term *relation* may refer either to a base relation or to a view, unless we explicitly specify one or the other. Privileges at the relation level specify for each user the individual relations on which each type of command can be applied. Some privileges also refer to individual columns (attributes) of relations. SQL2 commands provide privileges at the *relation and attribute level only.* Although this is quite general, it makes it difficult to create accounts with limited privileges. The granting and revoking of privileges generally follow an authorization model for discretionary privileges known as the **access matrix model**, where the rows of a matrix M represent *subjects* (users, accounts, programs) and the columns represent *objects* (relations, records, columns, views, operations). Each position $M(i, j)$ in the matrix represents the types of privileges (read, write, update) that subject i holds on object j.

To control the granting and revoking of relation privileges, each relation R in a database is assigned an **owner account**, which is typically the account that was used when the relation was created in the first place. The owner of a relation is given *all* privileges on that relation. In SQL2, the DBA can assign an owner to a whole schema by creating the schema and associating the appropriate authorization identifier with that schema, using the CREATE SCHEMA command (see Section 4.1.1). The owner account holder can pass privileges on any of the owned relations to other users by **granting** privileges to their accounts. In SQL the following types of privileges can be granted on each individual relation R:

- **SELECT (retrieval or read) privilege on *R*.** Gives the account retrieval privilege. In SQL this gives the account the privilege to use the SELECT statement to retrieve tuples from R.
- **Modification privileges on *R*.** This gives the account the capability to modify the tuples of R. In SQL this includes three privileges: UPDATE, DELETE, and INSERT. These correspond to the three SQL commands (see Section 4.4) for modifying a table R. Additionally, both the INSERT and UPDATE privileges can specify that only certain attributes of R can be modified by the account.

- **References privilege on *R*.** This gives the account the capability to *reference* (or refer to) a relation *R* when specifying integrity constraints. This privilege can also be restricted to specific attributes of *R*.

Notice that to create a view, the account must have the SELECT privilege on *all relations* involved in the view definition in order to specify the query that corresponds to the view.

24.2.2 Specifying Privileges through the Use of Views

The mechanism of **views** is an important *discretionary authorization mechanism* in its own right. For example, if the owner *A* of a relation *R* wants another account *B* to be able to retrieve only some fields of *R,* then *A* can create a view *V* of *R* that includes only those attributes and then grant SELECT on *V* to *B*. The same applies to limiting *B* to retrieving only certain tuples of *R;* a view *V′* can be created by defining the view by means of a query that selects only those tuples from *R* that *A* wants to allow *B* to access. We will illustrate this discussion with the example given in Section 24.2.5.

24.2.3 Revoking of Privileges

In some cases it is desirable to grant a privilege to a user temporarily. For example, the owner of a relation may want to grant the SELECT privilege to a user for a specific task and then revoke that privilege once the task is completed. Hence, a mechanism for **revoking** privileges is needed. In SQL a REVOKE command is included for the purpose of canceling privileges. We will see how the REVOKE command is used in the example in Section 24.2.5.

24.2.4 Propagation of Privileges Using the GRANT OPTION

Whenever the owner *A* of a relation *R* grants a privilege on *R* to another account *B*, the privilege can be given to *B with* or *without* the **GRANT OPTION**. If the GRANT OPTION is given, this means that *B* can also grant that privilege on *R* to other accounts. Suppose that *B* is given the GRANT OPTION by *A* and that *B* then grants the privilege on *R* to a third account *C*, also with the GRANT OPTION. In this way, privileges on *R* can **propagate** to other accounts without the knowledge of the owner of *R*. If the owner account *A* now revokes the privilege granted to *B*, all the privileges that *B* propagated based on that privilege *should automatically be revoked* by the system.

It is possible for a user to receive a certain privilege from two or more sources. For example, A4 may receive a certain UPDATE *R* privilege from *both* A2 and A3. In such a case, if A2 revokes this privilege from A4, A4 will still continue to have the privilege by virtue of having been granted it from A3. If A3 later revokes the privilege from A4, A4 totally loses the privilege. Hence, a DBMS that allows propagation of privileges must keep track of how all the privileges were granted so that revoking of privileges can be done correctly and completely.

24.2.5 An Example to Illustrate Granting and Revoking of Privileges

Suppose that the DBA creates four accounts—A1, A2, A3, and A4—and wants only A1 to be able to create base relations. To do this, the DBA must issue the following GRANT command in SQL:

```
GRANT CREATETAB TO A1;
```

The CREATETAB (create table) privilege gives account A1 the capability to create new database tables (base relations) and is hence an *account privilege.* This privilege was part of earlier versions of SQL but is now left to each individual system implementation to define.

In SQL2 the same effect can be accomplished by having the DBA issue a CREATE SCHEMA command, as follows:

```
CREATE SCHEMA EXAMPLE AUTHORIZATION A1;
```

User account A1 can now create tables under the schema called EXAMPLE. To continue our example, suppose that A1 creates the two base relations EMPLOYEE and DEPARTMENT shown in Figure 24.1; A1 is then the **owner** of these two relations and hence has *all the relation privileges* on each of them.

Next, suppose that account A1 wants to grant to account A2 the privilege to insert and delete tuples in both of these relations. However, A1 does not want A2 to be able to propagate these privileges to additional accounts. A1 can issue the following command:

```
GRANT INSERT, DELETE ON EMPLOYEE, DEPARTMENT TO A2;
```

Notice that the owner account A1 of a relation automatically has the GRANT OPTION, allowing it to grant privileges on the relation to other accounts. However, account A2 cannot grant INSERT and DELETE privileges on the EMPLOYEE and DEPARTMENT tables because A2 was not given the GRANT OPTION in the preceding command.

Next, suppose that A1 wants to allow account A3 to retrieve information from either of the two tables and also to be able to propagate the SELECT privilege to other accounts. A1 can issue the following command:

```
GRANT SELECT ON EMPLOYEE, DEPARTMENT TO A3 WITH GRANT OPTION;
```

EMPLOYEE

Name	Ssn	Bdate	Address	Sex	Salary	Dno

DEPARTMENT

Dnumber	Dname	Mgr_ssn

Figure 24.1
Schemas for the two relations EMPLOYEE and DEPARTMENT.

The clause WITH GRANT OPTION means that A3 can now propagate the privilege to other accounts by using GRANT. For example, A3 can grant the SELECT privilege on the EMPLOYEE relation to A4 by issuing the following command:

```
GRANT SELECT ON EMPLOYEE TO A4;
```

Notice that A4 cannot propagate the SELECT privilege to other accounts because the GRANT OPTION was not given to A4.

Now suppose that A1 decides to revoke the SELECT privilege on the EMPLOYEE relation from A3; A1 then can issue this command:

```
REVOKE SELECT ON EMPLOYEE FROM A3;
```

The DBMS must now revoke the SELECT privilege on EMPLOYEE from A3, and it must also *automatically revoke* the SELECT privilege on EMPLOYEE from A4. This is because A3 granted that privilege to A4, but A3 does not have the privilege any more.

Next, suppose that A1 wants to give back to A3 a limited capability to SELECT from the EMPLOYEE relation and wants to allow A3 to be able to propagate the privilege. The limitation is to retrieve only the Name, Bdate, and Address attributes and only for the tuples with Dno = 5. A1 then can create the following view:

```
CREATE VIEW A3EMPLOYEE AS
SELECT Name, Bdate, Address
FROM EMPLOYEE
WHERE Dno = 5;
```

After the view is created, A1 can grant SELECT on the view A3EMPLOYEE to A3 as follows:

```
GRANT SELECT ON A3EMPLOYEE TO A3 WITH GRANT OPTION;
```

Finally, suppose that A1 wants to allow A4 to update only the Salary attribute of EMPLOYEE; A1 can then issue the following command:

```
GRANT UPDATE ON EMPLOYEE (Salary) TO A4;
```

The UPDATE and INSERT privileges can specify particular attributes that may be updated or inserted in a relation. Other privileges (SELECT, DELETE) are not attribute specific, because this specificity can easily be controlled by creating the appropriate views that include only the desired attributes and granting the corresponding privileges on the views. However, because updating views is not always possible (see Chapter 5), the UPDATE and INSERT privileges are given the option to specify the particular attributes of a base relation that may be updated.

24.2.6 Specifying Limits on Propagation of Privileges

Techniques to limit the propagation of privileges have been developed, although they have not yet been implemented in most DBMSs and *are not a part* of SQL. Limiting **horizontal propagation** to an integer number i means that an account B given the GRANT OPTION can grant the privilege to at most i other accounts.

Vertical propagation is more complicated; it limits the depth of the granting of privileges. Granting a privilege with a vertical propagation of zero is equivalent to granting the privilege with *no* GRANT OPTION. If account *A* grants a privilege to account *B* with the vertical propagation set to an integer number $j > 0$, this means that the account *B* has the GRANT OPTION on that privilege, but *B* can grant the privilege to other accounts only with a vertical propagation *less than j.* In effect, vertical propagation limits the sequence of GRANT OPTIONS that can be given from one account to the next based on a single original grant of the privilege.

We briefly illustrate horizontal and vertical propagation limits—which are *not available* currently in SQL or other relational systems—with an example. Suppose that A1 grants SELECT to A2 on the EMPLOYEE relation with horizontal propagation equal to 1 and vertical propagation equal to 2. A2 can then grant SELECT to at most one account because the horizontal propagation limitation is set to 1. Additionally, A2 cannot grant the privilege to another account except with vertical propagation set to 0 (no GRANT OPTION) or 1; this is because A2 must reduce the vertical propagation by at least 1 when passing the privilege to others. In addition, the horizontal propagation must be less than or equal to the originally granted horizontal propagation. For example, if account *A* grants a privilege to account *B* with the horizontal propagation set to an integer number $j > 0$, this means that *B* can grant the privilege to other accounts only with a horizontal propagation *less than or equal to j.* As this example shows, horizontal and vertical propagation techniques are designed to limit the depth and breadth of propagation of privileges.

24.3 Mandatory Access Control and Role-Based Access Control for Multilevel Security

The discretionary access control technique of granting and revoking privileges on relations has traditionally been the main security mechanism for relational database systems. This is an all-or-nothing method: A user either has or does not have a certain privilege. In many applications, an *additional security policy* is needed that classifies data and users based on security classes. This approach, known as **mandatory access control** (**MAC**), would typically be *combined* with the discretionary access control mechanisms described in Section 24.2. It is important to note that most commercial DBMSs currently provide mechanisms only for discretionary access control. However, the need for multilevel security exists in government, military, and intelligence applications, as well as in many industrial and corporate applications. Some DBMS vendors—for example, Oracle—have released special versions of their RDBMSs that incorporate mandatory access control for government use.

Typical **security classes** are top secret (TS), secret (S), confidential (C), and unclassified (U), where TS is the highest level and U the lowest. Other more complex security classification schemes exist, in which the security classes are organized in a lattice. For simplicity, we will use the system with four security classification levels, where $TS \geq S \geq C \geq U$, to illustrate our discussion. The commonly used model for multilevel security, known as the *Bell-LaPadula model*, classifies each **subject** (user,

account, program) and **object** (relation, tuple, column, view, operation) into one of the security classifications TS, S, C, or U. We will refer to the **clearance** (classification) of a subject S as **class(*S*)** and to the **classification** of an object O as **class(*O*)**. Two restrictions are enforced on data access based on the subject/object classifications:

1. A subject S is not allowed read access to an object O unless class(S) ≥ class(O). This is known as the **simple security property**.
2. A subject S is not allowed to write an object O unless class(S) ≤ class(O). This is known as the **star property** (or *-property).

The first restriction is intuitive and enforces the obvious rule that no subject can read an object whose security classification is higher than the subject's security clearance. The second restriction is less intuitive. It prohibits a subject from writing an object at a lower security classification than the subject's security clearance. Violation of this rule would allow information to flow from higher to lower classifications, which violates a basic tenet of multilevel security. For example, a user (subject) with TS clearance may make a copy of an object with classification TS and then write it back as a new object with classification U, thus making it visible throughout the system.

To incorporate multilevel security notions into the relational database model, it is common to consider attribute values and tuples as data objects. Hence, each attribute A is associated with a **classification attribute** C in the schema, and each attribute value in a tuple is associated with a corresponding security classification. In addition, in some models, a **tuple classification** attribute TC is added to the relation attributes to provide a classification for each tuple as a whole. The model we describe here is known as the *multilevel model*, because it allows classifications at multiple security levels. A **multilevel relation** schema R with n attributes would be represented as:

$$R(A_1, C_1, A_2, C_2, ..., A_n, C_n, TC)$$

where each C_i represents the *classification attribute* associated with attribute A_i.

The value of the tuple classification attribute TC in each tuple t—which is the *highest* of all attribute classification values within t—provides a general classification for the tuple itself. Each attribute classification C_i provides a finer security classification for each attribute value within the tuple. The value of TC in each tuple t is the *highest* of all attribute classification values C_i within t.

The **apparent key** of a multilevel relation is the set of attributes that would have formed the primary key in a regular (single-level) relation. A multilevel relation will appear to contain different data to subjects (users) with different clearance levels. In some cases, it is possible to store a single tuple in the relation at a higher classification level and produce the corresponding tuples at a lower-level classification through a process known as **filtering**. In other cases, it is necessary to store two or more tuples at different classification levels with the same value for the *apparent key*.

This leads to the concept of **polyinstantiation**,[4] where several tuples can have the same apparent key value but have different attribute values for users at different clearance levels.

We illustrate these concepts with the simple example of a multilevel relation shown in Figure 24.2(a), where we display the classification attribute values next to each attribute's value. Assume that the Name attribute is the apparent key, and consider the query **SELECT** * **FROM** EMPLOYEE. A user with security clearance *S* would see the same relation shown in Figure 24.2(a), since all tuple classifications are less than or equal to *S*. However, a user with security clearance *C* would not be allowed to see the values for Salary of 'Brown' and Job_performance of 'Smith', since they have higher classification. The tuples would be *filtered* to appear as shown in Figure 24.2(b), with Salary and Job_performance *appearing as null.* For a user with security clearance *U*, the filtering allows only the Name attribute of 'Smith' to appear, with all the other

(a) **EMPLOYEE**

Name	Salary	JobPerformance	TC
Smith U	40000 C	Fair S	S
Brown C	80000 S	Good C	S

(b) **EMPLOYEE**

Name	Salary	JobPerformance	TC
Smith U	40000 C	NULL C	C
Brown C	NULL C	Good C	C

(c) **EMPLOYEE**

Name	Salary	JobPerformance	TC
Smith U	NULL U	NULL U	U

(d) **EMPLOYEE**

Name	Salary	JobPerformance	TC
Smith U	40000 C	Fair S	S
Smith U	40000 C	Excellent C	C
Brown C	80000 S	Good C	S

Figure 24.2
A multilevel relation to illustrate multilevel security. (a) The original EMPLOYEE tuples. (b) Appearance of EMPLOYEE after filtering for classification C users. (c) Appearance of EMPLOYEE after filtering for classification U users. (d) Polyinstantiation of the Smith tuple.

[4]This is similar to the notion of having multiple versions in the database that represent the same real-world object.

attributes appearing as null (Figure 24.2(c)). Thus, filtering introduces null values for attribute values whose security classification is higher than the user's security clearance.

In general, the **entity integrity** rule for multilevel relations states that all attributes that are members of the apparent key must not be null and must have the *same* security classification within each individual tuple. Additionally, all other attribute values in the tuple must have a security classification greater than or equal to that of the apparent key. This constraint ensures that a user can see the key if the user is permitted to see any part of the tuple. Other integrity rules, called **null integrity** and **interinstance integrity**, informally ensure that if a tuple value at some security level can be filtered (derived) from a higher-classified tuple, then it is sufficient to store the higher-classified tuple in the multilevel relation.

To illustrate polyinstantiation further, suppose that a user with security clearance *C* tries to update the value of Job_performance of 'Smith' in Figure 24.2 to 'Excellent'; this corresponds to the following SQL update being submitted by that user:

```
UPDATE   EMPLOYEE
SET      Job_performance = 'Excellent'
WHERE    Name = 'Smith';
```

Since the view provided to users with security clearance *C* (see Figure 24.2(b)) permits such an update, the system should not reject it; otherwise, the user could *infer* that some nonnull value exists for the Job_performance attribute of 'Smith' rather than the null value that appears. This is an example of inferring information through what is known as a **covert channel**, which should not be permitted in highly secure systems (see Section 24.6.1). However, the user should not be allowed to overwrite the existing value of Job_performance at the higher classification level. The solution is to create a **polyinstantiation** for the 'Smith' tuple at the lower classification level *C*, as shown in Figure 24.2(d). This is necessary since the new tuple cannot be filtered from the existing tuple at classification *S*.

The basic update operations of the relational model (INSERT, DELETE, UPDATE) must be modified to handle this and similar situations, but this aspect of the problem is outside the scope of our presentation. We refer the interested reader to the Selected Bibliography at the end of this chapter for further details.

24.3.1 Comparing Discretionary Access Control and Mandatory Access Control

Discretionary access control (DAC) policies are characterized by a high degree of flexibility, which makes them suitable for a large variety of application domains. The main drawback of DAC models is their vulnerability to malicious attacks, such as Trojan horses embedded in application programs. The reason is that discretionary authorization models do not impose any control on how information is propagated and used once it has been accessed by users authorized to do so. By contrast, mandatory policies ensure a high degree of protection—in a way, they prevent

any illegal flow of information. Therefore, they are suitable for military and high security types of applications, which require a higher degree of protection. However, mandatory policies have the drawback of being too rigid in that they require a strict classification of subjects and objects into security levels, and therefore they are applicable to few environments. In many practical situations, discretionary policies are preferred because they offer a better tradeoff between security and applicability.

24.3.2 Role-Based Access Control

Role-based access control (RBAC) emerged rapidly in the 1990s as a proven technology for managing and enforcing security in large-scale enterprise-wide systems. Its basic notion is that privileges and other permissions are associated with organizational **roles**, rather than individual users. Individual users are then assigned to appropriate roles. Roles can be created using the CREATE ROLE and DESTROY ROLE commands. The GRANT and REVOKE commands discussed in Section 24.2 can then be used to assign and revoke privileges from roles, as well as for individual users when needed. For example, a company may have roles such as sales account manager, purchasing agent, mailroom clerk, department manager, and so on. Multiple individuals can be assigned to each role. Security privileges that are common to a role are granted to the role name, and any individual assigned to this role would automatically have those privileges granted.

RBAC can be used with traditional discretionary and mandatory access controls; it ensures that only authorized users in their specified roles are given access to certain data or resources. Users create sessions during which they may activate a subset of roles to which they belong. Each session can be assigned to several roles, but it maps to one user or a single subject only. Many DBMSs have allowed the concept of roles, where privileges can be assigned to roles.

Separation of duties is another important requirement in various commercial DBMSs. It is needed to prevent one user from doing work that requires the involvement of two or more people, thus preventing collusion. One method in which separation of duties can be successfully implemented is with mutual exclusion of roles. Two roles are said to be **mutually exclusive** if both the roles cannot be used simultaneously by the user. **Mutual exclusion of roles** can be categorized into two types, namely *authorization time exclusion (static)* and *runtime exclusion (dynamic)*. In authorization time exclusion, two roles that have been specified as mutually exclusive cannot be part of a user's authorization at the same time. In runtime exclusion, both these roles can be authorized to one user but cannot be activated by the user at the same time. Another variation in mutual exclusion of roles is that of complete and partial exclusion.

The **role hierarchy** in RBAC is a natural way to organize roles to reflect the organization's lines of authority and responsibility. By convention, junior roles at the bottom are connected to progressively senior roles as one moves up the hierarchy. The hierarchic diagrams are partial orders, so they are reflexive, transitive, and

antisymmetric. In other words, if a user has one role, the user automatically has roles lower in the hierarchy. Defining a role hierarchy involves choosing the type of hierarchy and the roles, and then implementing the hierarchy by granting roles to other roles. Role hierarchy can be implemented in the following manner:

```
GRANT ROLE full_time TO employee_type1
GRANT ROLE intern TO employee_type2
```

The above are examples of granting the roles *full_time* and *intern* to two types of employees.

Another issue related to security is *identity management*. **Identity** refers to a unique name of an individual person. Since the legal names of persons are not necessarily unique, the identity of a person must include sufficient additional information to make the complete name unique. Authorizing this identity and managing the schema of these identities is called **Identity Management.** Identity Management addresses how organizations can effectively authenticate people and manage their access to confidential information. It has become more visible as a business requirement across all industries affecting organizations of all sizes. Identity Management administrators constantly need to satisfy application owners while keeping expenditures under control and increasing IT efficiency.

Another important consideration in RBAC systems is the possible temporal constraints that may exist on roles, such as the time and duration of role activations, and timed triggering of a role by an activation of another role. Using an RBAC model is a highly desirable goal for addressing the key security requirements of Web-based applications. Roles can be assigned to workflow tasks so that a user with any of the roles related to a task may be authorized to execute it and may play a certain role only for a certain duration.

RBAC models have several desirable features, such as flexibility, policy neutrality, better support for security management and administration, and other aspects that make them attractive candidates for developing secure Web-based applications. These features are lacking in DAC and MAC models. In addition, RBAC models include the capabilities available in traditional DAC and MAC policies. Furthermore, an RBAC model provides mechanisms for addressing the security issues related to the execution of tasks and workflows, and for specifying user-defined and organization-specific policies. Easier deployment over the Internet has been another reason for the success of RBAC models.

24.3.3 Label-Based Security and Row-Level Access Control

Many commercial DBMSs currently use the concept of row-level access control, where sophisticated access control rules can be implemented by considering the data row by row. In row-level access control, each data row is given a label, which is used to store information about data sensitivity. Row-level access control provides finer granularity of data security by allowing the permissions to be set for each row and not just for the table or column. Initially the user is given a default session label by the database administrator. Levels correspond to a hierarchy of data-sensitivity

levels to exposure or corruption, with the goal of maintaining privacy or security. Labels are used to prevent unauthorized users from viewing or altering certain data. A user having a low authorization level, usually represented by a low number, is denied access to data having a higher-level number. If no such label is given to a row, a row label is automatically assigned to it depending upon the user's session label.

A policy defined by an administrator is called a **Label Security policy.** Whenever data affected by the policy is accessed or queried through an application, the policy is automatically invoked. When a policy is implemented, a new column is added to each row in the schema. The added column contains the label for each row that reflects the sensitivity of the row as per the policy. Similar to MAC, where each user has a security clearance, each user has an identity in label-based security. This user's identity is compared to the label assigned to each row to determine whether the user has access to view the contents of that row. However, the user can write the label value himself, within certain restrictions and guidelines for that specific row. This label can be set to a value that is between the user's current session label and the user's minimum level. The DBA has the privilege to set an initial default row label.

The Label Security requirements are applied on top of the DAC requirements for each user. Hence, the user must satisfy the DAC requirements and then the label security requirements to access a row. The DAC requirements make sure that the user is legally authorized to carry on that operation on the schema. In most applications, only some of the tables need label-based security. For the majority of the application tables, the protection provided by DAC is sufficient.

Security policies are generally created by managers and human resources personnel. The policies are high-level, technology neutral, and relate to risks. Policies are a result of management instructions to specify organizational procedures, guiding principles, and courses of action that are considered to be expedient, prudent, or advantageous. Policies are typically accompanied by a definition of penalties and countermeasures if the policy is transgressed. These policies are then interpreted and converted to a set of label-oriented policies by the **Label Security administrator**, who defines the security labels for data and authorizations for users; these labels and authorizations govern access to specified protected objects.

Suppose a user has SELECT privileges on a table. When the user executes a SELECT statement on that table, Label Security will automatically evaluate each row returned by the query to determine whether the user has rights to view the data. For example, if the user has a sensitivity of 20, then the user can view all rows having a security level of 20 or lower. The level determines the sensitivity of the information contained in a row; the more sensitive the row, the higher its security label value. Such Label Security can be configured to perform security checks on UPDATE, DELETE, and INSERT statements as well.

24.3.4 XML Access Control

With the worldwide use of XML in commercial and scientific applications, efforts are under way to develop security standards. Among these efforts are digital

signatures and encryption standards for XML. The XML Signature Syntax and Processing specification describes an XML syntax for representing the associations between cryptographic signatures and XML documents or other electronic resources. The specification also includes procedures for computing and verifying XML signatures. An XML digital signature differs from other protocols for message signing, such as **PGP** (**Pretty Good Privacy**—a confidentiality and authentication service that can be used for electronic mail and file storage application), in its support for signing only specific portions of the XML tree (see Chapter 12) rather than the complete document. Additionally, the XML signature specification defines mechanisms for countersigning and transformations—so-called *canonicalization* to ensure that two instances of the same text produce the same digest for signing even if their representations differ slightly, for example, in typographic white space.

The XML Encryption Syntax and Processing specification defines XML vocabulary and processing rules for protecting confidentiality of XML documents in whole or in part and of non-XML data as well. The encrypted content and additional processing information for the recipient are represented in well-formed XML so that the result can be further processed using XML tools. In contrast to other commonly used technologies for confidentiality such as SSL (Secure Sockets Layer—a leading Internet security protocol), and virtual private networks, XML encryption also applies to parts of documents and to documents in persistent storage.

24.3.5 Access Control Policies for E-Commerce and the Web

Electronic commerce (**e-commerce**) environments are characterized by any transactions that are done electronically. They require elaborate access control policies that go beyond traditional DBMSs. In conventional database environments, access control is usually performed using a set of authorizations stated by security officers or users according to some security policies. Such a simple paradigm is not well suited for a dynamic environment like e-commerce. Furthermore, in an e-commerce environment the resources to be protected are not only traditional data but also knowledge and experience. Such peculiarities call for more flexibility in specifying access control policies. The access control mechanism must be flexible enough to support a wide spectrum of heterogeneous protection objects.

A second related requirement is the support for content-based access control. **Content-based access control** allows one to express access control policies that take the protection object content into account. In order to support content-based access control, access control policies must allow inclusion of conditions based on the object content.

A third requirement is related to the heterogeneity of subjects, which requires access control policies based on user characteristics and qualifications rather than on specific and individual characteristics (for example, user IDs). A possible solution, to better take into account user profiles in the formulation of access control policies, is to support the notion of credentials. A **credential** is a set of properties concerning a user that are relevant for security purposes (for example, age or position or role

within an organization). For instance, by using credentials, one can simply formulate policies such as *Only permanent staff with five or more years of service can access documents related to the internals of the system.*

It is believed that the XML is expected to play a key role in access control for e-commerce applications[5] because XML is becoming the common representation language for document interchange over the Web, and is also becoming the language for e-commerce. Thus, on the one hand there is the need to make XML representations secure, by providing access control mechanisms specifically tailored to the protection of XML documents. On the other hand, access control information (that is, access control policies and user credentials) can be expressed using XML itself. The **Directory Services Markup Language** (DSML) is a representation of directory service information in XML syntax. It provides a foundation for a standard for communicating with the directory services that will be responsible for providing and authenticating user credentials. The uniform presentation of both protection objects and access control policies can be applied to policies and credentials themselves. For instance, some credential properties (such as the user name) may be accessible to everyone, whereas other properties may be visible only to a restricted class of users. Additionally, the use of an XML-based language for specifying credentials and access control policies facilitates secure credential submission and export of access control policies.

24.4 SQL Injection

SQL Injection is one of the most common threats to a database system. We will discuss it in detail later in this section. Some of the other attacks on databases that are quite frequent are:

- **Unauthorized privilege escalation.** This attack is characterized by an individual attempting to elevate his or her privilege by attacking vulnerable points in the database systems.
- **Privilege abuse.** While the previous attack is done by an unauthorized user, this attack is performed by a privileged user. For example, an administrator who is allowed to change student information can use this privilege to update student grades without the instructor's permission.
- **Denial of service.** A **Denial of Service (DOS) attack** is an attempt to make resources unavailable to its intended users. It is a general attack category in which access to network applications or data is denied to intended users by overflowing the buffer or consuming resources.
- **Weak Authentication.** If the user authentication scheme is weak, an attacker can impersonate the identity of a legitimate user by obtaining their login credentials.

[5]See Thuraisingham et al. (2001).

24.4.1 SQL Injection Methods

As we discussed in Chapter 14, Web programs and applications that access a database can send commands and data to the database, as well as display data retrieved from the database through the Web browser. In an **SQL Injection attack**, the attacker injects a string input through the application, which changes or manipulates the SQL statement to the attacker's advantage. An SQL Injection attack can harm the database in various ways, such as unauthorized manipulation of the database, or retrieval of sensitive data. It can also be used to execute system level commands that may cause the system to deny service to the application. This section describes types of injection attacks.

SQL Manipulation. A manipulation attack, which is the most common type of injection attack, changes an SQL command in the application—for example, by adding conditions to the WHERE-clause of a query, or by expanding a query with additional query components using set operations such as UNION, INTERSECT, or MINUS. Other types of manipulation attacks are also possible. A typical manipulation attack occurs during database login. For example, suppose that a simplistic authentication procedure issues the following query and checks to see if any rows were returned:

SELECT * **FROM** users **WHERE** username = 'jake' and PASSWORD = 'jakespasswd'.

The attacker can try to change (or manipulate) the SQL statement, by changing it as follows:

SELECT * **FROM** users **WHERE** username = 'jake' and (PASSWORD = 'jakespasswd' or 'x' = 'x')

As a result, the attacker who knows that 'jake' is a valid login of some user is able to log into the database system as 'jake' without knowing his password and is able to do everything that 'jake' may be authorized to do to the database system.

Code Injection. This type of attack attempts to add additional SQL statements or commands to the existing SQL statement by exploiting a computer bug, which is caused by processing invalid data. The attacker can inject or introduce code into a computer program to change the course of execution. Code injection is a popular technique for system hacking or cracking to gain information.

Function Call Injection. In this kind of attack, a database function or operating system function call is inserted into a vulnerable SQL statement to manipulate the data or make a privileged system call. For example, it is possible to exploit a function that performs some aspect related to network communication. In addition, functions that are contained in a customized database package, or any custom database function, can be executed as part of an SQL query. In particular, dynamically created SQL queries (see Chapter 13) can be exploited since they are constructed at run time.

For example, the *dual* table is used in the FROM clause of SQL in Oracle when a user needs to run SQL that does not logically have a table name. To get today's date, we can use:

```
SELECT SYSDATE FROM dual;
```

The following example demonstrates that even the simplest SQL statements can be vulnerable.

```
SELECT TRANSLATE ('user input', 'from_string', 'to_string') FROM dual;
```

Here, TRANSLATE is used to replace a string of characters with another string of characters. The TRANSLATE function above will replace the characters of the 'from_string' with the characters in the 'to_string' one by one. This means that the *f* will be replaced with the *t*, the *r* with the *o*, the *o* with the _, and so on.

This type of SQL statement can be subjected to a function injection attack. Consider the following example:

```
SELECT TRANSLATE (" || UTL_HTTP.REQUEST ('http://129.107.2.1/') || ",
    '98765432', '9876') FROM dual;
```

The user can input the string (" || UTL_HTTP.REQUEST ('http://129.107.2.1/') || "), where || is the concatenate operator, thus requesting a page from a Web server. UTL_HTTP makes Hypertext Transfer Protocol (HTTP) callouts from SQL. The REQUEST object takes a URL ('http://129.107.2.1/' in this example) as a parameter, contacts that site, and returns the data (typically HTML) obtained from that site. The attacker could manipulate the string he inputs, as well as the URL, to include other functions and do other illegal operations. We just used a dummy example to show conversion of '98765432' to '9876', but the user's intent would be to access the URL and get sensitive information. The attacker can then retrieve useful information from the database server—located at the URL that is passed as a parameter—and send it to the Web server (that calls the TRANSLATE function).

24.4.2 Risks Associated with SQL Injection

SQL injection is harmful and the risks associated with it provide motivation for attackers. Some of the risks associated with SQL injection attacks are explained below.

- **Database Fingerprinting.** The attacker can determine the type of database being used in the backend so that he can use database-specific attacks that correspond to weaknesses in a particular DBMS.
- **Denial of Service.** The attacker can flood the server with requests, thus denying service to valid users, or they can delete some data.
- **Bypassing Authentication.** This is one of the most common risks, in which the attacker can gain access to the database as an authorized user and perform all the desired tasks.

- **Identifying Injectable Parameters.** In this type of attack, the attacker gathers important information about the type and structure of the back-end database of a Web application. This attack is made possible by the fact that the default error page returned by application servers is often overly descriptive.
- **Executing Remote Commands.** This provides attackers with a tool to execute arbitrary commands on the database. For example, a remote user can execute stored database procedures and functions from a remote SQL interactive interface.
- **Performing Privilege Escalation.** This type of attack takes advantage of logical flaws within the database to upgrade the access level.

24.4.3 Protection Techniques against SQL Injection

Protection against SQL injection attacks can be achieved by applying certain programming rules to all Web-accessible procedures and functions. This section describes some of these techniques.

Bind Variables (Using Parameterized Statements). The use of bind variables (also known as *parameters*; see Chapter 13) protects against injection attacks and also improves performance.

Consider the following example using Java and JDBC:

```
PreparedStatement stmt = conn.prepareStatement( "SELECT * FROM
    EMPLOYEE WHERE EMPLOYEE_ID=? AND PASSWORD=?");
stmt.setString(1, employee_id);
stmt.setString(2, password);
```

Instead of embedding the user input into the statement, the input should be bound to a parameter. In this example, the input '1' is assigned (bound) to a bind variable 'employee_id' and input '2' to the bind variable 'password' instead of directly passing string parameters.

Filtering Input (Input Validation). This technique can be used to remove escape characters from input strings by using the SQL `Replace` function. For example, the delimiter single quote (') can be replaced by two single quotes (''). Some SQL Manipulation attacks can be prevented by using this technique, since escape characters can be used to inject manipulation attacks. However, because there can be a large number of escape characters, this technique is not reliable.

Function Security. Database functions, both standard and custom, should be restricted, as they can be exploited in the SQL function injection attacks.

24.5 Introduction to Statistical Database Security

Statistical databases are used mainly to produce statistics about various populations. The database may contain confidential data about individuals, which should be protected from user access. However, users are permitted to retrieve statistical information about the populations, such as averages, sums, counts, maximums, minimums, and standard deviations. The techniques that have been developed to protect the privacy of individual information are beyond the scope of this book. We will illustrate the problem with a very simple example, which refers to the relation shown in Figure 24.3. This is a PERSON relation with the attributes Name, Ssn, Income, Address, City, State, Zip, Sex, and Last_degree.

A **population** is a set of tuples of a relation (table) that satisfy some selection condition. Hence, each selection condition on the PERSON relation will specify a particular population of PERSON tuples. For example, the condition Sex = 'M' specifies the male population; the condition ((Sex = 'F') AND (Last_degree = 'M.S.' OR Last_degree = 'Ph.D.')) specifies the female population that has an M.S. or Ph.D. degree as their highest degree; and the condition City = 'Houston' specifies the population that lives in Houston.

Statistical queries involve applying statistical functions to a population of tuples. For example, we may want to retrieve the number of individuals in a population or the average income in the population. However, statistical users are not allowed to retrieve individual data, such as the income of a specific person. **Statistical database security** techniques must prohibit the retrieval of individual data. This can be achieved by prohibiting queries that retrieve attribute values and by allowing only queries that involve statistical aggregate functions such as COUNT, SUM, MIN, MAX, AVERAGE, and STANDARD DEVIATION. Such queries are sometimes called **statistical queries**.

It is the responsibility of a database management system to ensure the confidentiality of information about individuals, while still providing useful statistical summaries of data about those individuals to users. Provision of **privacy protection** of users in a statistical database is paramount; its violation is illustrated in the following example.

In some cases it is possible to **infer** the values of individual tuples from a sequence of statistical queries. This is particularly true when the conditions result in a

PERSON

Name	Ssn	Income	Address	City	State	Zip	Sex	Last_degree

Figure 24.3
The PERSON relation schema for illustrating statistical database security.

population consisting of a small number of tuples. As an illustration, consider the following statistical queries:

Q1: **SELECT COUNT** (*) **FROM** PERSON
WHERE <condition>;

Q2: **SELECT AVG** (Income) **FROM** PERSON
WHERE <condition>;

Now suppose that we are interested in finding the Salary of Jane Smith, and we know that she has a Ph.D. degree and that she lives in the city of Bellaire, Texas. We issue the statistical query Q1 with the following condition:

(Last_degree='Ph.D.' AND Sex='F' AND City='Bellaire' AND State='Texas')

If we get a result of 1 for this query, we can issue Q2 with the same condition and find the Salary of Jane Smith. Even if the result of Q1 on the preceding condition is not 1 but is a small number—say 2 or 3—we can issue statistical queries using the functions MAX, MIN, and AVERAGE to identify the possible range of values for the Salary of Jane Smith.

The possibility of inferring individual information from statistical queries is reduced if no statistical queries are permitted whenever the number of tuples in the population specified by the selection condition falls below some threshold. Another technique for prohibiting retrieval of individual information is to prohibit sequences of queries that refer repeatedly to the same population of tuples. It is also possible to introduce slight inaccuracies or *noise* into the results of statistical queries deliberately, to make it difficult to deduce individual information from the results. Another technique is partitioning of the database. Partitioning implies that records are stored in groups of some minimum size; queries can refer to any complete group or set of groups, but never to subsets of records within a group. The interested reader is referred to the bibliography at the end of this chapter for a discussion of these techniques.

24.6 Introduction to Flow Control

Flow control regulates the distribution or flow of information among accessible objects. A flow between object *X* and object *Y* occurs when a program reads values from *X* and writes values into *Y*. **Flow controls** check that information contained in some objects does not flow explicitly or implicitly into less protected objects. Thus, a user cannot get indirectly in *Y* what he or she cannot get directly in *X*. Active flow control began in the early 1970s. Most flow controls employ some concept of security class; the transfer of information from a sender to a receiver is allowed only if the receiver's security class is at least as privileged as the sender's. Examples of a flow control include preventing a service program from leaking a customer's confidential data, and blocking the transmission of secret military data to an unknown classified user.

A **flow policy** specifies the channels along which information is allowed to move. The simplest flow policy specifies just two classes of information—confidential (*C*)

and nonconfidential (*N*)—and allows all flows except those from class *C* to class *N*. This policy can solve the confinement problem that arises when a service program handles data such as customer information, some of which may be confidential. For example, an income-tax computing service might be allowed to retain a customer's address and the bill for services rendered, but not a customer's income or deductions.

Access control mechanisms are responsible for checking users' authorizations for resource access: Only granted operations are executed. Flow controls can be enforced by an extended access control mechanism, which involves assigning a security class (usually called the *clearance*) to each running program. The program is allowed to read a particular memory segment only if its security class is as high as that of the segment. It is allowed to write in a segment only if its class is as low as that of the segment. This automatically ensures that no information transmitted by the person can move from a higher to a lower class. For example, a military program with a secret clearance can only read from objects that are unclassified and confidential and can only write into objects that are secret or top secret.

Two types of flow can be distinguished: *explicit flows,* occurring as a consequence of assignment instructions, such as $Y{:=}\, f(X_1, X_n,)$, and *implicit flows* generated by conditional instructions, such as if $f(X_{m+1,\ldots,} X_n)$ then $Y{:=}\, f\,(X_1, X_m)$.

Flow control mechanisms must verify that only authorized flows, both explicit and implicit, are executed. A set of rules must be satisfied to ensure secure information flows. Rules can be expressed using flow relations among classes and assigned to information, stating the authorized flows within a system. (An information flow from *A* to *B* occurs when information associated with *A* affects the value of information associated with *B*. The flow results from operations that cause information transfer from one object to another.) These relations can define, for a class, the set of classes where information (classified in that class) can flow, or can state the specific relations to be verified between two classes to allow information to flow from one to the other. In general, flow control mechanisms implement the controls by assigning a label to each object and by specifying the security class of the object. Labels are then used to verify the flow relations defined in the model.

24.6.1 Covert Channels

A covert channel allows a transfer of information that violates the security or the policy. Specifically, a **covert channel** allows information to pass from a higher classification level to a lower classification level through improper means. Covert channels can be classified into two broad categories: timing channels and storage. The distinguishing feature between the two is that in a **timing channel** the information is conveyed by the timing of events or processes, whereas **storage channels** do not require any temporal synchronization, in that information is conveyed by accessing system information or what is otherwise inaccessible to the user.

In a simple example of a covert channel, consider a distributed database system in which two nodes have user security levels of secret (S) and unclassified (U). In order

for a transaction to commit, both nodes must agree to commit. They mutually can only do operations that are consistent with the *-property, which states that in any transaction, the *S* site cannot write or pass information to the *U* site. However, if these two sites collude to set up a covert channel between them, a transaction involving secret data may be committed unconditionally by the *U* site, but the *S* site may do so in some predefined agreed-upon way so that certain information may be passed from the *S* site to the *U* site, violating the *-property. This may be achieved where the transaction runs repeatedly, but the actions taken by the *S* site implicitly convey information to the *U* site. Measures such as locking, which we discussed in Chapters 22 and 23, prevent concurrent writing of the information by users with different security levels into the same objects, preventing the storage-type covert channels. Operating systems and distributed databases provide control over the multiprogramming of operations that allows a sharing of resources without the possibility of encroachment of one program or process into another's memory or other resources in the system, thus preventing timing-oriented covert channels. In general, covert channels are not a major problem in well-implemented robust database implementations. However, certain schemes may be contrived by clever users that implicitly transfer information.

Some security experts believe that one way to avoid covert channels is to disallow programmers to actually gain access to sensitive data that a program will process after the program has been put into operation. For example, a programmer for a bank has no need to access the names or balances in depositors' accounts. Programmers for brokerage firms do not need to know what buy and sell orders exist for clients. During program testing, access to a form of real data or some sample test data may be justifiable, but not after the program has been accepted for regular use.

24.7 Encryption and Public Key Infrastructures

The previous methods of access and flow control, despite being strong control measures, may not be able to protect databases from some threats. Suppose we communicate data, but our data falls into the hands of a nonlegitimate user. In this situation, by using encryption we can disguise the message so that even if the transmission is diverted, the message will not be revealed. **Encryption** is the conversion of data into a form, called a **ciphertext**, which cannot be easily understood by unauthorized persons. It enhances security and privacy when access controls are bypassed, because in cases of data loss or theft, encrypted data cannot be easily understood by unauthorized persons.

With this background, we adhere to following standard definitions:[6]

- *Ciphertext*: Encrypted (enciphered) data.

[6]These definitions are from NIST (National Institute of Standards and Technology) from http://csrc.nist.gov/publications/nistpubs/800-67/SP800-67.pdf.

- *Plaintext (or cleartext)*: Intelligible data that has meaning and can be read or acted upon without the application of decryption.
- *Encryption*: The process of transforming plaintext into ciphertext.
- *Decryption*: The process of transforming ciphertext back into plaintext.

Encryption consists of applying an **encryption algorithm** to data using some prespecified **encryption key**. The resulting data has to be **decrypted** using a **decryption key** to recover the original data.

24.7.1 The Data Encryption and Advanced Encryption Standards

The **Data Encryption Standard** (DES) is a system developed by the U.S. government for use by the general public. It has been widely accepted as a cryptographic standard both in the United States and abroad. DES can provide end-to-end encryption on the channel between sender *A* and receiver *B*. The DES algorithm is a careful and complex combination of two of the fundamental building blocks of encryption: substitution and permutation (transposition). The algorithm derives its strength from repeated application of these two techniques for a total of 16 cycles. Plaintext (the original form of the message) is encrypted as blocks of 64 bits. Although the key is 64 bits long, in effect the key can be any 56-bit number. After questioning the adequacy of DES, the NIST introduced the **Advanced Encryption Standard** (AES). This algorithm has a block size of 128 bits, compared with DES's 56-block size, and can use keys of 128, 192, or 256 bits, compared with DES's 56-bit key. AES introduces more possible keys, compared with DES, and thus takes a much longer time to crack.

24.7.2 Symmetric Key Algorithms

A symmetric key is one key that is used for both encryption and decryption. By using a symmetric key, fast encryption and decryption is possible for routine use with sensitive data in the database. A message encrypted with a secret key can be decrypted only with the same secret key. Algorithms used for symmetric key encryption are called **secret-key algorithms**. Since secret-key algorithms are mostly used for encrypting the content of a message, they are also called **content-encryption algorithms**.

The major liability associated with secret-key algorithms is the need for sharing the secret key. A possible method is to derive the secret key from a user-supplied password string by applying the same function to the string at both the sender and receiver; this is known as a *password-based encryption algorithm.* The strength of the symmetric key encryption depends on the size of the key used. For the same algorithm, encrypting using a longer key is tougher to break than the one using a shorter key.

24.7.3 Public (Asymmetric) Key Encryption

In 1976, Diffie and Hellman proposed a new kind of cryptosystem, which they called **public key encryption**. Public key algorithms are based on mathematical

functions rather than operations on bit patterns. They address one drawback of symmetric key encryption, namely that both sender and recipient must exchange the common key in a secure manner. In public key systems, two keys are used for encryption/decryption. The *public key* can be transmitted in a non-secure way, whereas the *private key* is not transmitted at all. These algorithms—which use two related keys, a public key and a private key, to perform complementary operations (encryption and decryption)—are known as **asymmetric key encryption algorithms**. The use of two keys can have profound consequences in the areas of confidentiality, key distribution, and authentication. The two keys used for public key encryption are referred to as the **public key** and the **private key**. The private key is kept secret, but it is referred to as a *private key* rather than a *secret key* (the key used in conventional encryption) to avoid confusion with conventional encryption. The two keys are mathematically related, since one of the keys is used to perform encryption and the other to perform decryption. However, it is very difficult to derive the private key from the public key.

A public key encryption scheme, or *infrastructure*, has six ingredients:

1. **Plaintext.** This is the data or readable message that is fed into the algorithm as input.
2. **Encryption algorithm.** This algorithm performs various transformations on the plaintext.
3. and 4. **Public and private keys.** These are a pair of keys that have been selected so that if one is used for encryption, the other is used for decryption. The exact transformations performed by the encryption algorithm depend on the public or private key that is provided as input. For example, if a message is encrypted using the public key, it can only be decrypted using the private key.
5. **Ciphertext.** This is the scrambled message produced as output. It depends on the plaintext and the key. For a given message, two different keys will produce two different ciphertexts.
6. **Decryption algorithm.** This algorithm accepts the ciphertext and the matching key and produces the original plaintext.

As the name suggests, the public key of the pair is made public for others to use, whereas the private key is known only to its owner. A general-purpose public key cryptographic algorithm relies on one key for encryption and a different but related key for decryption. The essential steps are as follows:

1. Each user generates a pair of keys to be used for the encryption and decryption of messages.
2. Each user places one of the two keys in a public register or other accessible file. This is the public key. The companion key is kept private.
3. If a sender wishes to send a private message to a receiver, the sender encrypts the message using the receiver's public key.

4. When the receiver receives the message, he or she decrypts it using the receiver's private key. No other recipient can decrypt the message because only the receiver knows his or her private key.

The RSA Public Key Encryption Algorithm. One of the first public key schemes was introduced in 1978 by Ron Rivest, Adi Shamir, and Len Adleman at MIT and is named after them as the **RSA scheme**. The RSA scheme has since then reigned supreme as the most widely accepted and implemented approach to public key encryption. The RSA encryption algorithm incorporates results from number theory, combined with the difficulty of determining the prime factors of a target. The RSA algorithm also operates with modular arithmetic—mod n.

Two keys, d and e, are used for decryption and encryption. An important property is that they can be interchanged. n is chosen as a large integer that is a product of two large distinct prime numbers, a and b, $n = a \times b$. The encryption key e is a randomly chosen number between 1 and n that is relatively prime to $(a - 1) \times (b - 1)$. The plaintext block P is encrypted as P^e where $P^e = P \bmod n$. Because the exponentiation is performed mod n, factoring P^e to uncover the encrypted plaintext is difficult. However, the decrypting key d is carefully chosen so that $(P^e)^d \bmod n = P$. The decryption key d can be computed from the condition that $d \times e = 1 \bmod ((a - 1) \times (b - 1))$. Thus, the legitimate receiver who knows d simply computes $(P^e)^d \bmod n = P$ and recovers P without having to factor P^e.

24.7.4 Digital Signatures

A digital signature is an example of using encryption techniques to provide authentication services in electronic commerce applications. Like a handwritten signature, a **digital signature** is a means of associating a mark unique to an individual with a body of text. The mark should be unforgettable, meaning that others should be able to check that the signature comes from the originator.

A digital signature consists of a string of symbols. If a person's digital signature were always the same for each message, then one could easily counterfeit it by simply copying the string of symbols. Thus, signatures must be different for each use. This can be achieved by making each digital signature a function of the message that it is signing, together with a timestamp. To be unique to each signer and counterfeitproof, each digital signature must also depend on some secret number that is unique to the signer. Thus, in general, a counterfeitproof digital signature must depend on the message and a unique secret number of the signer. The verifier of the signature, however, should not need to know any secret number. Public key techniques are the best means of creating digital signatures with these properties.

24.7.5 Digital Certificates

A digital certificate is used to combine the value of a public key with the identity of the person or service that holds the corresponding private key into a digitally signed

statement. Certificates are issued and signed by a certification authority (CA). The entity receiving this certificate from a CA is the subject of that certificate. Instead of requiring each participant in an application to authenticate every user, third-party authentication relies on the use of digital certificates.

The digital certificate itself contains various types of information. For example, both the certification authority and the certificate owner information are included. The following list describes all the information included in the certificate:

1. The certificate owner information, which is represented by a unique identifier known as the distinguished name (DN) of the owner. This includes the owner's name, as well as the owner's organization and other information about the owner.
2. The certificate also includes the public key of the owner.
3. The date of issue of the certificate is also included.
4. The validity period is specified by 'Valid From' and 'Valid To' dates, which are included in each certificate.
5. Issuer identifier information is included in the certificate.
6. Finally, the digital signature of the issuing CA for the certificate is included. All the information listed is encoded through a message-digest function, which creates the digital signature. The digital signature basically certifies that the association between the certificate owner and public key is valid.

24.8 Privacy Issues and Preservation

Preserving data privacy is a growing challenge for database security and privacy experts. In some perspectives, to preserve data privacy we should even limit performing large-scale data mining and analysis. The most commonly used techniques to address this concern are to avoid building mammoth central warehouses as a single repository of vital information. Another possible measure is to intentionally modify or perturb data.

If all data were available at a single warehouse, violating only a single repository's security could expose all data. Avoiding central warehouses and using distributed data mining algorithms minimizes the exchange of data needed to develop globally valid models. By modifying, perturbing, and anonymizing data, we can also mitigate privacy risks associated with data mining. This can be done by removing identity information from the released data and injecting noise into the data. However, by using these techniques, we should pay attention to the quality of the resulting data in the database, which may undergo too many modifications. We must be able to estimate the errors that may be introduced by these modifications.

Privacy is an important area of ongoing research in database management. It is complicated due to its multidisciplinary nature and the issues related to the subjectivity in the interpretation of privacy, trust, and so on. As an example, consider medical and legal records and transactions, which must maintain certain privacy

requirements while they are being defined and enforced. Providing access control and privacy for mobile devices is also receiving increased attention. DBMSs need robust techniques for efficient storage of security-relevant information on small devices, as well as trust negotiation techniques. Where to keep information related to user identities, profiles, credentials, and permissions and how to use it for reliable user identification remains an important problem. Because large-sized streams of data are generated in such environments, efficient techniques for access control must be devised and integrated with processing techniques for continuous queries. Finally, the privacy of user location data, acquired from sensors and communication networks, must be ensured.

24.9 Challenges of Database Security

Considering the vast growth in volume and speed of threats to databases and information assets, research efforts need to be devoted to the following issues: data quality, intellectual property rights, and database survivability. These are only some of the main challenges that researchers in database security are trying to address.

24.9.1 Data Quality

The database community needs techniques and organizational solutions to assess and attest the quality of data. These techniques may include simple mechanisms such as quality stamps that are posted on Web sites. We also need techniques that provide more effective integrity semantics verification and tools for the assessment of data quality, based on techniques such as record linkage. Application-level recovery techniques are also needed for automatically repairing incorrect data. The ETL (extract, transform, load) tools widely used to load data in data warehouses (see Section 29.4) are presently grappling with these issues.

24.9.2 Intellectual Property Rights

With the widespread use of the Internet and intranets, legal and informational aspects of data are becoming major concerns of organizations. To address these concerns, watermarking techniques for relational data have been proposed. The main purpose of digital watermarking is to protect content from unauthorized duplication and distribution by enabling provable ownership of the content. It has traditionally relied upon the availability of a large noise domain within which the object can be altered while retaining its essential properties. However, research is needed to assess the robustness of such techniques and to investigate different approaches aimed at preventing intellectual property rights violations.

24.9.3 Database Survivability

Database systems need to operate and continue their functions, even with reduced capabilities, despite disruptive events such as information warfare attacks. A DBMS,

in addition to making every effort to prevent an attack and detecting one in the event of occurrence, should be able to do the following:

- **Confinement.** Take immediate action to eliminate the attacker's access to the system and to isolate or contain the problem to prevent further spread.
- **Damage assessment.** Determine the extent of the problem, including failed functions and corrupted data.
- **Reconfiguration.** Reconfigure to allow operation to continue in a degraded mode while recovery proceeds.
- **Repair.** Recover corrupted or lost data and repair or reinstall failed system functions to reestablish a normal level of operation.
- **Fault treatment.** To the extent possible, identify the weaknesses exploited in the attack and take steps to prevent a recurrence.

The goal of the information warfare attacker is to damage the organization's operation and fulfillment of its mission through disruption of its information systems. The specific target of an attack may be the system itself or its data. While attacks that bring the system down outright are severe and dramatic, they must also be well timed to achieve the attacker's goal, since attacks will receive immediate and concentrated attention in order to bring the system back to operational condition, diagnose how the attack took place, and install preventive measures.

To date, issues related to database survivability have not been sufficiently investigated. Much more research needs to be devoted to techniques and methodologies that ensure database system survivability.

24.10 Oracle Label-Based Security

Restricting access to entire tables or isolating sensitive data into separate databases is a costly operation to administer. **Oracle Label Security** overcomes the need for such measures by enabling row-level access control. It is available in Oracle Database 11g Release 1 (11.1) Enterprise Edition at the time of writing. Each database table or view has a security policy associated with it. This policy executes every time the table or view is queried or altered. Developers can readily add label-based access control to their Oracle Database applications. Label-based security provides an adaptable way of controlling access to sensitive data. Both users and data have labels associated with them. Oracle Label Security uses these labels to provide security.

24.10.1 Virtual Private Database (VPD) Technology

Virtual Private Databases (VPDs) is a feature of the Oracle Enterprise Edition that adds predicates to user statements to limit their access in a transparent manner to the user and the application. The VPD concept allows server-enforced, fine-grained access control for a secure application.

VPD provides access control based on policies. These VPD policies enforce object-level access control or row-level security. It provides an application programming

interface (API) that allows security policies to be attached to database tables or views. Using PL/SQL, a host programming language used in Oracle applications, developers and security administrators can implement security policies with the help of stored procedures.[7] VPD policies allow developers to remove access security mechanisms from applications and centralize them within the Oracle Database.

VPD is enabled by associating a security "policy" with a table, view, or synonym. An administrator uses the supplied PL/SQL package, DBMS_RLS, to bind a policy function with a database object. When an object having a security policy associated with it is accessed, the function implementing this policy is consulted. The policy function returns a predicate (a WHERE clause) which is then appended to the user's SQL statement, thus *transparently* and *dynamically* modifying the user's data access. Oracle Label Security is a technique of enforcing row-level security in the form of a security policy.

24.10.2 Label Security Architecture

Oracle Label Security is built on the VPD technology delivered in the Oracle Database 11.1 Enterprise Edition. Figure 24.4 illustrates how data is accessed under Oracle Label Security, showing the sequence of DAC and label security checks.

Figure 24.4 shows the sequence of discretionary access control (DAC) and label security checks. The left part of the figure shows an application user in an Oracle Database 11g Release 1 (11.1) session sending out an SQL request. The Oracle DBMS checks the DAC privileges of the user, making sure that he or she has SELECT privileges on the table. Then it checks whether the table has a Virtual Private Database (VPD) policy associated with it to determine if the table is protected using Oracle Label Security. If it is, the VPD SQL modification (WHERE clause) is added to the original SQL statement to find the set of accessible rows for the user to view. Then Oracle Label Security checks the labels on each row, to determine the subset of rows to which the user has access (as explained in the next section). This modified query gets processed, optimized, and executed.

24.10.3 How Data Labels and User Labels Work Together

A user's label indicates the information the user is permitted to access. It also determines the type of access (read or write) that the user has on that information. A row's label shows the sensitivity of the information that the row contains as well as the ownership of the information. When a table in the database has a label-based access associated with it, a row can be accessed only if the user's label meet certain criteria defined in the policy definitions. Access is granted or denied based on the result of comparing the data label and the session label of the user.

Compartments allow a finer classification of sensitivity of the labeled data. All data related to the same project can be labeled with the same compartment. Compartments are optional; a label can contain zero or more compartments.

[7] Stored procedures are discussed in Section 5.2.2.

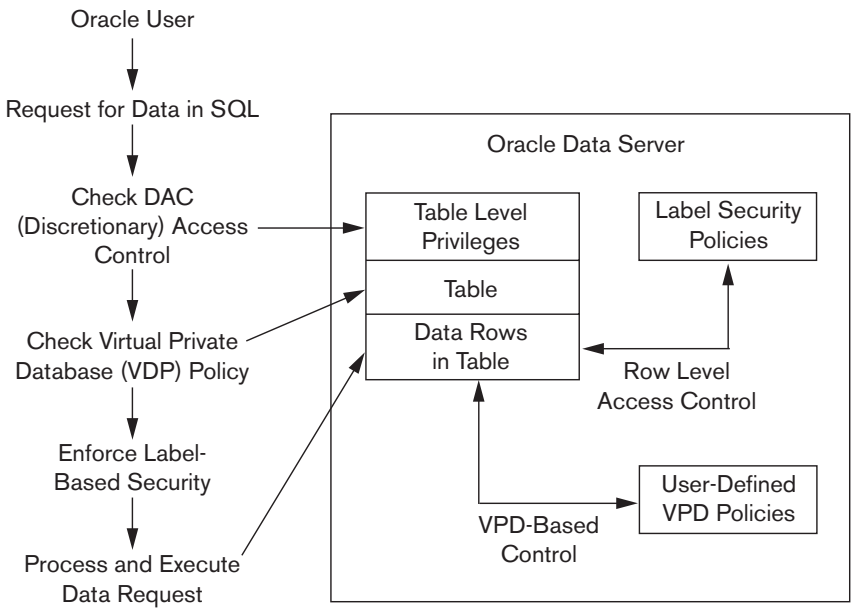


Figure 24.4
Oracle Label Security architecture.
Source: Oracle (2007)

Groups are used to identify organizations as owners of the data with corresponding group labels. Groups are hierarchical; for example, a group can be associated with a parent group.

If a user has a maximum level of SENSITIVE, then the user potentially has access to all data having levels SENSITIVE, CONFIDENTIAL, and UNCLASSIFIED. This user has no access to HIGHLY_SENSITIVE data. Figure 24.5 shows how data labels and user labels work together to provide access control in Oracle Label Security.

As shown in Figure 24.5, User 1 can access the rows 2, 3, and 4 because his maximum level is HS (Highly_Sensitive). He has access to the FIN (Finance) compartment, and his access to group WR (Western Region) hierarchically includes group WR_SAL (WR Sales). He cannot access row 1 because he does not have the CHEM (Chemical) compartment. It is important that a user has authorization for all compartments in a row's data label to be able to access that row. Based on this example, user 2 can access both rows 3 and 4, and has a maximum level of S, which is less than the HS in row 2. So, although user 2 has access to the FIN compartment, he can only access the group WR_SAL, and thus cannot acces row 1.

24.11 Summary

In this chapter we discussed several techniques for enforcing database system security. We presented different threats to databases in terms of loss of integrity, availability, and confidentiality. We discussed the types of control measures to deal with these problems: access control, inference control, flow control, and encryption. In

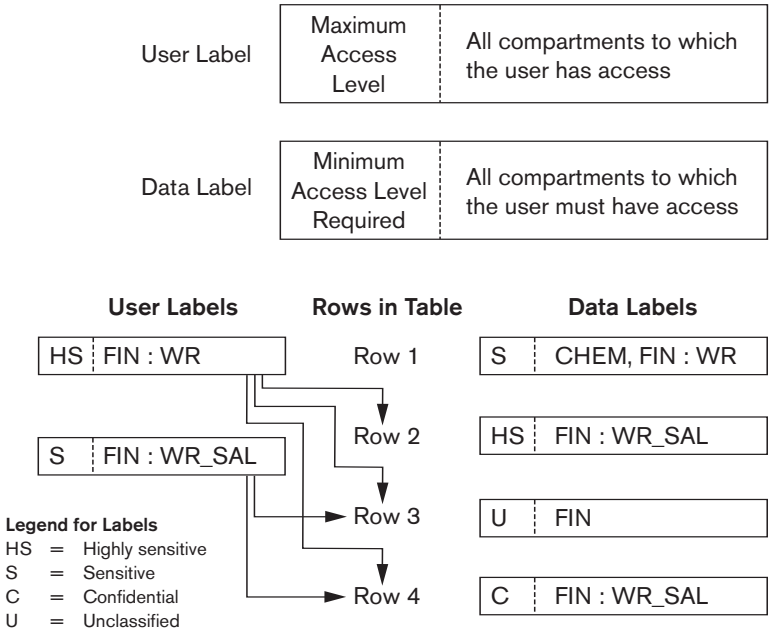


Figure 24.5
Data labels and user labels in Oracle.
Source: Oracle (2007)

the introduction we covered various issues related to security including data sensitivity and type of disclosures, providing security vs. precision in the result when a user requests information, and the relationship between information security and privacy.

Security enforcement deals with controlling access to the database system as a whole and controlling authorization to access specific portions of a database. The former is usually done by assigning accounts with passwords to users. The latter can be accomplished by using a system of granting and revoking privileges to individual accounts for accessing specific parts of the database. This approach is generally referred to as discretionary access control (DAC). We presented some SQL commands for granting and revoking privileges, and we illustrated their use with examples. Then we gave an overview of mandatory access control (MAC) mechanisms that enforce multilevel security. These require the classifications of users and data values into security classes and enforce the rules that prohibit flow of information from higher to lower security levels. Some of the key concepts underlying the multilevel relational model, including filtering and polyinstantiation, were presented. Role-based access control (RBAC) was introduced, which assigns privileges based on roles that users play. We introduced the notion of role hierarchies, mutual exclusion of roles, and row- and label-based security. We briefly discussed the problem of controlling access to statistical databases to protect the privacy of individual information while concurrently providing statistical access to populations of records. We explained the main ideas behind the threat of SQL Injection, the methods in which it can be induced, and the various types of risks associated with it. Then we gave an

idea of the various ways SQL injection can be prevented. The issues related to flow control and the problems associated with covert channels were discussed next, as well as encryption and public-private key-based infrastructures. The idea of symmetric key algorithms and the use of the popular asymmetric key-based public key infrastructure (PKI) scheme was explained. We also covered the concepts of digital signatures and digital certificates. We highlighted the importance of privacy issues and hinted at some privacy preservation techniques. We discussed a variety of challenges to security including data quality, intellectual property rights, and data survivability. We ended the chapter by introducing the implementation of security policies by using a combination of label-based security and virtual private databases in Oracle 11g.

Review Questions

24.1. Discuss what is meant by each of the following terms: *database authorization, access control, data encryption, privileged (system) account, database audit, audit trail.*

24.2. Which account is designated as the owner of a relation? What privileges does the owner of a relation have?

24.3. How is the view mechanism used as an authorization mechanism?

24.4. Discuss the types of privileges at the account level and those at the relation level.

24.5. What is meant by granting a privilege? What is meant by revoking a privilege?

24.6. Discuss the system of propagation of privileges and the restraints imposed by horizontal and vertical propagation limits.

24.7. List the types of privileges available in SQL.

24.8. What is the difference between *discretionary* and *mandatory* access control?

24.9. What are the typical security classifications? Discuss the simple security property and the *-property, and explain the justification behind these rules for enforcing multilevel security.

24.10. Describe the multilevel relational data model. Define the following terms: *apparent key, polyinstantiation, filtering.*

24.11. What are the relative merits of using DAC or MAC?

24.12. What is role-based access control? In what ways is it superior to DAC and MAC?

24.13. What are the two types of mutual exclusion in role-based access control?

24.14. What is meant by row-level access control?

24.15. What is label security? How does an administrator enforce it?

24.16. What are the different types of SQL injection attacks?

24.17. What risks are associated with SQL injection attacks?

24.18. What preventive measures are possible against SQL injection attacks?

24.19. What is a statistical database? Discuss the problem of statistical database security.

24.20. How is privacy related to statistical database security? What measures can be taken to ensure some degree of privacy in statistical databases?

24.21. What is flow control as a security measure? What types of flow control exist?

24.22. What are covert channels? Give an example of a covert channel.

24.23. What is the goal of encryption? What process is involved in encrypting data and then recovering it at the other end?

24.24. Give an example of an encryption algorithm and explain how it works.

24.25. Repeat the previous question for the popular RSA algorithm.

24.26. What is a symmetric key algorithm for key-based security?

24.27. What is the public key infrastructure scheme? How does it provide security?

24.28. What are digital signatures? How do they work?

24.29. What type of information does a digital certificate include?

Exercises

24.30. How can privacy of data be preserved in a database?

24.31. What are some of the current outstanding challenges for database security?

24.32. Consider the relational database schema in Figure 3.5. Suppose that all the relations were created by (and hence are owned by) user *X*, who wants to grant the following privileges to user accounts *A*, *B*, *C*, *D*, and *E*:

a. Account *A* can retrieve or modify any relation except DEPENDENT and can grant any of these privileges to other users.

b. Account *B* can retrieve all the attributes of EMPLOYEE and DEPARTMENT except for Salary, Mgr_ssn, and Mgr_start_date.

c. Account *C* can retrieve or modify WORKS_ON but can only retrieve the Fname, Minit, Lname, and Ssn attributes of EMPLOYEE and the Pname and Pnumber attributes of PROJECT.

d. Account *D* can retrieve any attribute of EMPLOYEE or DEPENDENT and can modify DEPENDENT.

e. Account *E* can retrieve any attribute of EMPLOYEE but only for EMPLOYEE tuples that have Dno = 3.

f. Write SQL statements to grant these privileges. Use views where appropriate.

24.33. Suppose that privilege (a) of Exercise 24.32 is to be given with GRANT OPTION but only so that account *A* can grant it to at most five accounts, and each of these accounts can propagate the privilege to other accounts but *without* the GRANT OPTION privilege. What would the horizontal and vertical propagation limits be in this case?

24.34. Consider the relation shown in Figure 24.2(d). How would it appear to a user with classification *U*? Suppose that a classification *U* user tries to update the salary of 'Smith' to $50,000; what would be the result of this action?

Selected Bibliography

Authorization based on granting and revoking privileges was proposed for the SYSTEM R experimental DBMS and is presented in Griffiths and Wade (1976). Several books discuss security in databases and computer systems in general, including the books by Leiss (1982a) and Fernandez et al. (1981), and Fugini et al. (1995). Natan (2005) is a practical book on security and auditing implementation issues in all major RDBMSs.

Many papers discuss different techniques for the design and protection of statistical databases. They include McLeish (1989), Chin and Ozsoyoglu (1981), Leiss (1982), Wong (1984), and Denning (1980). Ghosh (1984) discusses the use of statistical databases for quality control. There are also many papers discussing cryptography and data encryption, including Diffie and Hellman (1979), Rivest et al. (1978), Akl (1983), Pfleeger and Pfleeger (2007), Omura et al. (1990), Stallings (2000), and Iyer at al. (2004).

Halfond et al. (2006) helps understand the concepts of SQL injection attacks and the various threats imposed by them. The white paper Oracle (2007a) explains how Oracle is less prone to SQL injection attack as compared to SQL Server. It also gives a brief explanation as to how these attacks can be prevented from occurring. Further proposed frameworks are discussed in Boyd and Keromytis (2004), Halfond and Orso (2005), and McClure and Krüger (2005).

Multilevel security is discussed in Jajodia and Sandhu (1991), Denning et al. (1987), Smith and Winslett (1992), Stachour and Thuraisingham (1990), Lunt et al. (1990), and Bertino et al. (2001). Overviews of research issues in database security are given by Lunt and Fernandez (1990), Jajodia and Sandhu (1991), Bertino (1998), Castano et al. (1995), and Thuraisingham et al. (2001). The effects of multilevel security on concurrency control are discussed in Atluri et al. (1997). Security in next-generation, semantic, and object-oriented databases is discussed in Rabbiti et al. (1991), Jajodia and Kogan (1990), and Smith (1990). Oh (1999) presents a model for both discretionary and mandatory security. Security models for Web-based applications and role-based access control are discussed in Joshi et al. (2001). Security issues for managers in the context of e-commerce applications and the need for risk assessment models for selection of appropriate security control measures are discussed in

Farahmand et al. (2005). Row-level access control is explained in detail in Oracle (2007b) and Sybase (2005). The latter also provides details on role hierarchy and mutual exclusion. Oracle (2009) explains how Oracle uses the concept of identity management.

Recent advances as well as future challenges for security and privacy of databases are discussed in Bertino and Sandhu (2005). U.S. Govt. (1978), OECD (1980), and NRC (2003) are good references on the view of privacy by important government bodies. Karat et al. (2009) discusses a policy framework for security and privacy. XML and access control are discussed in Naedele (2003). More details can be found on privacy preserving techniques in Vaidya and Clifton (2004), intellectual property rights in Sion et al. (2004), and database survivability in Jajodia et al. (1999). Oracle's VPD technology and label-based security is discussed in more detail in Oracle (2007b).

chapter 25

Distributed Databases

In this chapter we turn our attention to distributed databases (DDBs), distributed database management systems (DDBMSs), and how the client-server architecture is used as a platform for database application development. Distributed databases bring the advantages of distributed computing to the database management domain. A **distributed computing system** consists of a number of processing elements, not necessarily homogeneous, that are interconnected by a computer network, and that cooperate in performing certain assigned tasks. As a general goal, distributed computing systems partition a big, unmanageable problem into smaller pieces and solve it efficiently in a coordinated manner. The economic viability of this approach stems from two reasons: more computing power is harnessed to solve a complex task, and each autonomous processing element can be managed independently to develop its own applications.

DDB technology resulted from a merger of two technologies: database technology, and network and data communication technology. Computer networks allow distributed processing of data. Traditional databases, on the other hand, focus on providing centralized, controlled access to data. Distributed databases allow an integration of information and its processing by applications that may themselves be centralized or distributed.

Several distributed database prototype systems were developed in the 1980s to address the issues of data distribution, distributed query and transaction processing, distributed database metadata management, and other topics. However, a full-scale comprehensive DDBMS that implements the functionality and techniques proposed in DDB research never emerged as a commercially viable product. Most major vendors redirected their efforts from developing a *pure* DDBMS product into developing systems based on client-server concepts, or toward developing technologies for accessing distributed heterogeneous data sources.

Organizations continue to be interested in the *decentralization* of processing (at the system level) while achieving an *integration* of the information resources (at the logical level) within their geographically distributed systems of databases, applications, and users. There is now a general endorsement of the client-server approach to application development, and the three-tier approach to Web applications development (see Section 2.5).

In this chapter we discuss distributed databases, their architectural variations, and concepts central to data distribution and the management of distributed data. Details of the advances in communication technologies facilitating the development of DDBs are outside the scope of this book; see the texts on data communications and networking listed in the Selected Bibliography at the end of this chapter.

Section 25.1 introduces distributed database management and related concepts. Sections 25.2 and 25.3 introduce different types of distributed database systems and their architectures, including federated and multidatabase systems. The problems of heterogeneity and the needs of autonomy in federated database systems are also highlighted. Detailed issues of distributed database design, involving fragmenting of data and distributing it over multiple sites with possible replication, are discussed in Section 25.4. Sections 25.5 and 25.6 introduce distributed database query and transaction processing techniques, respectively. Section 25.7 gives an overview of the concurrency control and recovery in distributed databases. Section 25.8 discusses catalog management schemes in distributed databases. In Section 25.9, we briefly discuss current trends in distributed databases such as cloud computing and peer-to-peer databases. Section 25.10 discusses distributed database features of the Oracle RDBMS. Section 25.11 summarizes the chapter.

For a short introduction to the topic of distributed databases, Sections 25.1, 25.2, and 25.3 may be covered.

25.1 Distributed Database Concepts[1]

We can define a **distributed database (DDB)** as a collection of multiple logically interrelated databases distributed over a computer network, and a **distributed database management system (DDBMS)** as a software system that manages a distributed database while making the distribution transparent to the user.[2]

Distributed databases are different from Internet Web files. Web pages are basically a very large collection of files stored on different nodes in a network—the Internet—with interrelationships among the files represented via hyperlinks. The common functions of database management, including uniform query processing and transaction processing, *do not* apply to this scenario yet. The technology is, however, moving in a direction such that distributed World Wide Web (WWW) databases will become a reality in the future. We have discussed some of the issues of

[1]The substantial contribution of Narasimhan Srinivasan to this and several other sections in this chapter is appreciated.

[2]This definition and discussions in this section are based largely on Ozsu and Valduriez (1999).

accessing databases on the Web in Chapters 12 and 14. The proliferation of data at millions of Websites in various forms does *not* qualify as a DDB by the definition given earlier.

25.1.1 Differences between DDB and Multiprocessor Systems

We need to distinguish distributed databases from multiprocessor systems that use shared storage (primary memory or disk). For a database to be called distributed, the following minimum conditions should be satisfied:

- **Connection of database nodes over a computer network.** There are multiple computers, called **sites** or **nodes**. These sites must be connected by an underlying **communication network** to transmit data and commands among sites, as shown later in Figure 25.3(c).
- **Logical interrelation of the connected databases.** It is essential that the information in the databases be logically related.
- **Absence of homogeneity constraint among connected nodes.** It is not necessary that all nodes be identical in terms of data, hardware, and software.

The sites may all be located in physical proximity—say, within the same building or a 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** or **wide area network**. Local area networks typically use wireless hubs or cables, whereas long-haul networks use telephone lines or satellites. It is also possible to use a combination of networks.

Networks may have different **topologies** that define the direct communication paths among sites. The type and topology of the network used may have a significant impact on the performance and hence on the strategies for distributed query processing and distributed database design. For high-level architectural issues, however, it does not matter what type of network is used; what matters is that each site be 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 any particular topology. We will not address any network-specific issues, although it is important to understand that for an efficient operation of a distributed database system (DDBS), network design and performance issues are critical and are an integral part of the overall solution. The details of the underlying communication network are invisible to the end user.

25.1.2 Transparency

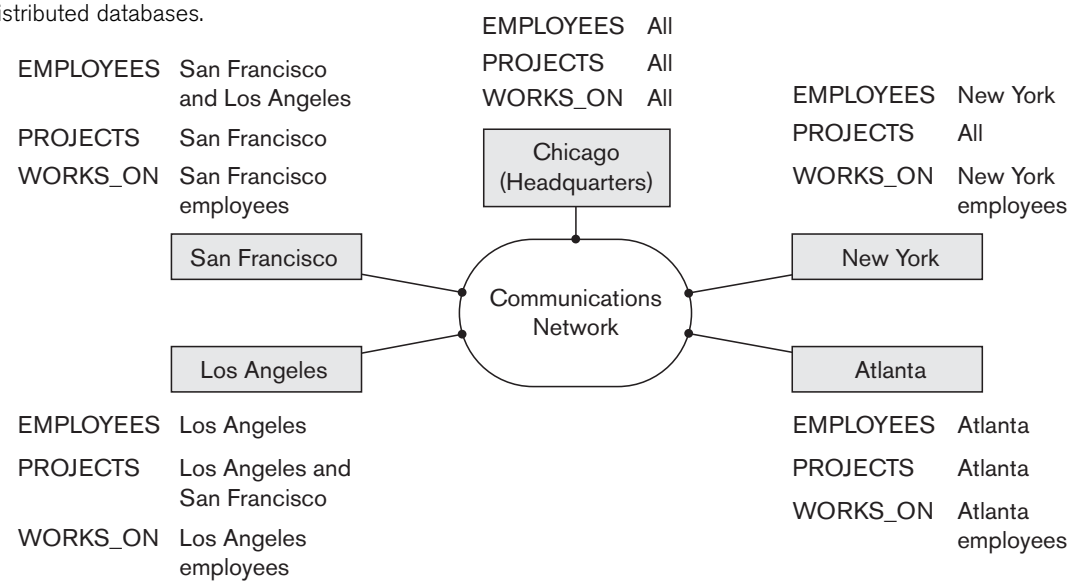
The concept of transparency extends the general idea of hiding implementation details from end users. A highly transparent system offers a lot of flexibility to the end user/application developer since it requires little or no awareness of underlying details on their part. In the case of a traditional centralized database, transparency simply pertains to logical and physical data independence for application developers. However, in a DDB scenario, the data and software are distributed over multiple

sites connected by a computer network, so additional types of transparencies are introduced.

Consider the company database in Figure 3.5 that we have been discussing throughout the book. The EMPLOYEE, PROJECT, and WORKS_ON tables may be fragmented horizontally (that is, into sets of rows, as we will discuss in Section 25.4) and stored with possible replication as shown in Figure 25.1. The following types of transparencies are possible:

- **Data organization transparency (also known as *distribution* or *network transparency*).** This refers to freedom for the user from the operational details of the network and the placement of the data in the distributed system. It may be divided into location transparency and naming transparency. **Location transparency** refers to the fact that the command used to perform a task is independent of the location of the data and the location of the node where the command was issued. **Naming transparency** implies that once a name is associated with an object, the named objects can be accessed unambiguously without additional specification as to where the data is located.
- **Replication transparency.** As we show in Figure 25.1, copies of the same data objects may be stored at multiple sites for better availability, performance, and reliability. Replication transparency makes the user unaware of the existence of these copies.
- **Fragmentation transparency.** Two types of fragmentation are possible. **Horizontal fragmentation** distributes a relation (table) into subrelations

Figure 25.1
Data distribution and replication among distributed databases.



that are subsets of the tuples (rows) in the original relation. **Vertical fragmentation** distributes a relation into subrelations where each subrelation is defined by a subset of the columns of the original relation. A global query by the user must be transformed into several fragment queries. Fragmentation transparency makes the user unaware of the existence of fragments.

- Other transparencies include **design transparency** and **execution transparency**—referring to freedom from knowing how the distributed database is designed and where a transaction executes.

25.1.3 Autonomy

Autonomy determines the extent to which individual nodes or DBs in a connected DDB can operate independently. A high degree of autonomy is desirable for increased flexibility and customized maintenance of an individual node. Autonomy can be applied to design, communication, and execution. **Design autonomy** refers to independence of data model usage and transaction management techniques among nodes. **Communication autonomy** determines the extent to which each node can decide on sharing of information with other nodes. **Execution autonomy** refers to independence of users to act as they please.

25.1.4 Reliability and Availability

Reliability and availability are two of the most common potential advantages cited for distributed databases. **Reliability** is broadly defined as the probability that a system is running (not down) at a certain time point, whereas **availability** is the probability that the system is continuously available during a time interval. We can directly relate reliability and availability of the database to the faults, errors, and failures associated with it. A failure can be described as a deviation of a system's behavior from that which is specified in order to ensure correct execution of operations. **Errors** constitute that subset of system states that causes the failure. **Fault** is the cause of an error.

To construct a system that is reliable, we can adopt several approaches. One common approach stresses *fault tolerance*; it recognizes that faults will occur, and designs mechanisms that can detect and remove faults before they can result in a system failure. Another more stringent approach attempts to ensure that the final system does not contain any faults. This is done through an exhaustive design process followed by extensive quality control and testing. A reliable DDBMS tolerates failures of underlying components and processes user requests so long as database consistency is not violated. A DDBMS recovery manager has to deal with failures arising from transactions, hardware, and communication networks. Hardware failures can either be those that result in loss of main memory contents or loss of secondary storage contents. Communication failures occur due to errors associated with messages and line failures. Message errors can include their loss, corruption, or out-of-order arrival at destination.

25.1.5 Advantages of Distributed Databases

Organizations resort to distributed database management for various reasons. Some important advantages are listed below.

1. **Improved ease and flexibility of application development**. Developing and maintaining applications at geographically distributed sites of an organization is facilitated owing to transparency of data distribution and control.
2. **Increased reliability and availability.** This is achieved by the isolation of faults to their site of origin without affecting the other databases connected to the network. When the data and DDBMS 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. In a distributed database, some of the data may be unreachable, but users may still be able to access other parts of the database. If the data in the failed site had been replicated at another site prior to the failure, then the user will not be affected at all.
3. **Improved performance.** A distributed DBMS fragments the database by keeping the data closer to where it is needed most. **Data localization** reduces the contention for CPU and I/O services and simultaneously reduces access delays involved in wide area networks. 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. Moreover, interquery and intraquery parallelism can be achieved by executing multiple queries at different sites, or by breaking up a query into a number of subqueries that execute in parallel. This contributes to improved performance.
4. **Easier expansion**. In a distributed environment, expansion of the system in terms of adding more data, increasing database sizes, or adding more processors is much easier.

The transparencies we discussed in Section 25.1.2 lead to a compromise between ease of use and the overhead cost of providing transparency. Total transparency provides the global user with a view of the entire DDBS as if it is a single centralized system. Transparency is provided as a complement to **autonomy**, which gives the users tighter control over local databases. Transparency features may be implemented as a part of the user language, which may translate the required services into appropriate operations. Additionally, transparency impacts the features that must be provided by the operating system and the DBMS.

25.1.6 Additional Functions of Distributed Databases

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 functions in addition to those of a centralized DBMS:

- **Keeping track of data distribution.** The ability to keep track of the data distribution, fragmentation, and replication by expanding the DDBMS catalog.
- **Distributed query processing.** The ability to access remote sites and transmit queries and data among the various sites via a communication network.
- **Distributed transaction management.** The ability to devise execution strategies for queries and transactions that access data from more than one site and to synchronize the access to distributed data and maintain the integrity of the overall database.
- **Replicated data management.** The ability to decide which copy of a replicated data item to access and to maintain the consistency of copies of a replicated data item.
- **Distributed database recovery.** The ability to recover from individual site crashes and from new types of failures, such as the failure of communication links.
- **Security.** Distributed transactions must be executed with the proper management of the security of the data and the authorization/access privileges of users.
- **Distributed directory (catalog) management.** A directory contains information (metadata) about data in the database. The directory may be global for the entire DDB, or local for each site. The placement and distribution of the directory are design and policy issues.

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 design issues and problems. Including all this additional functionality is hard to accomplish, and finding optimal solutions is a step beyond that.

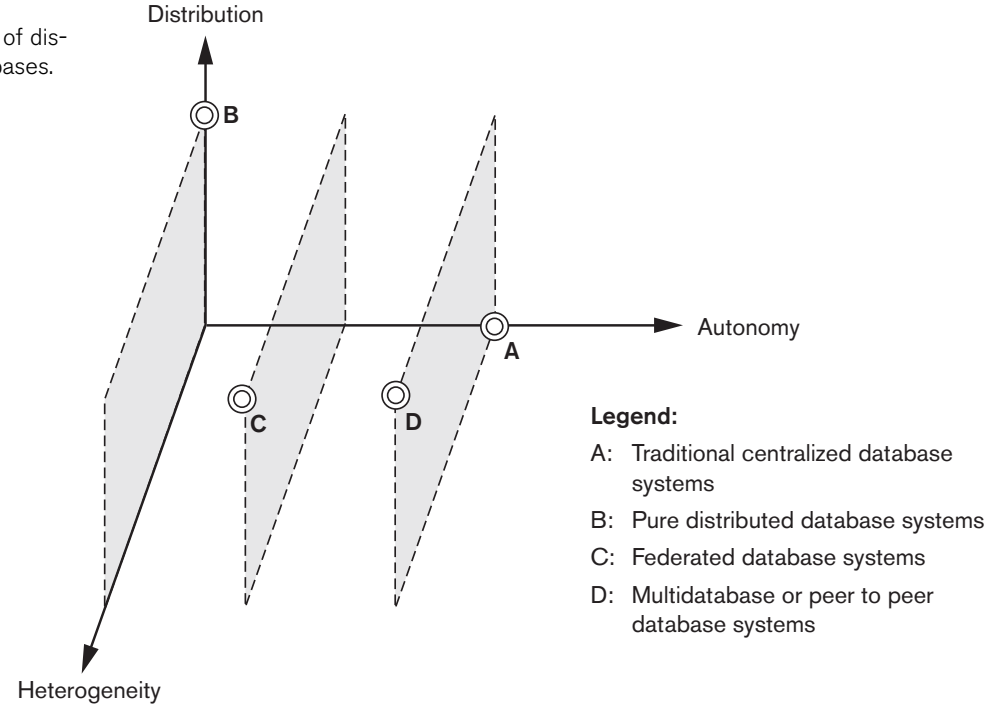
25.2 Types of Distributed Database Systems

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 users (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 there is no provision for the local site to function as a standalone DBMS, 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.

Figure 25.2 shows classification of DDBMS alternatives along orthogonal axes of distribution, autonomy, and heterogeneity. For a centralized database, there is complete autonomy, but a total lack of distribution and heterogeneity (Point A in the figure). We see that the degree of local autonomy provides further ground for classification into federated and multidatabase systems. At one extreme of the autonomy spectrum, we have a DDBMS that *looks like* a centralized DBMS to the user, with zero autonomy (Point B). A single conceptual schema exists, and all access to the system is obtained through a site that is part of the DDBMS—which means that no local autonomy exists. Along the autonomy axis we encounter two types of DDBMSs called *federated database system* (Point C) and *multidatabase system* (Point D). In such systems, each server is an independent and autonomous centralized DBMS that has its own local users, local transactions, and DBA, and hence has

Figure 25.2
Classification of distributed databases.



a very high degree of *local autonomy.* The term **federated database system (FDBS)** is used when there is some global view or schema of the federation of databases that is shared by the applications (Point C). On the other hand, a **multidatabase system** has full local autonomy in that it does not have a global schema but interactively constructs one as needed by the application (Point D).[3] Both systems are hybrids between distributed and centralized systems, and the distinction we made between them is not strictly followed. We will refer to them as FDBSs in a generic sense. Point D in the diagram may also stand for a system with full local autonomy and full heterogeneity—this could be a peer-to-peer database system (see Section 25.9.2). In a heterogeneous FDBS, one server may be a relational DBMS, another a network DBMS (such as Computer Associates' IDMS or HP'S IMAGE/3000), and a third an object DBMS (such as Object Design's ObjectStore) or hierarchical DBMS (such as IBM's IMS); in such a case, it is necessary to have a canonical system language and to include language translators to translate subqueries from the canonical language to the language of each server.

We briefly discuss the issues affecting the design of FDBSs next.

25.2.1 Federated Database Management Systems Issues

The type of heterogeneity present in FDBSs may arise from several sources. We discuss these sources first and then point out how the different types of autonomies contribute to a semantic heterogeneity that must be resolved in a heterogeneous FDBS.

- **Differences in data models.** Databases in an organization come from a variety of data models, including the so-called legacy models (hierarchical and network, see Web Appendixes D and E), the relational data model, the object data model, and even files. The modeling capabilities of the models vary. Hence, to deal with them uniformly via a single global schema or to process them in a single language is challenging. Even if two databases are both from the RDBMS environment, the same information may be represented as an attribute name, as a relation name, or as a value in different databases. This calls for an intelligent query-processing mechanism that can relate information based on metadata.
- **Differences in constraints.** Constraint facilities for specification and implementation vary from system to system. There are comparable features that must be reconciled in the construction of a global schema. For example, the relationships from ER models are represented as referential integrity constraints in the relational model. Triggers may have to be used to implement certain constraints in the relational model. The global schema must also deal with potential conflicts among constraints.

[3]The term *multidatabase system* is not easily applicable to most enterprise IT environments. The notion of constructing a global schema as and when the need arises is not very feasible in practice for enterprise databases.

- **Differences in query languages.** Even with the same data model, the languages and their versions vary. For example, SQL has multiple versions like SQL-89, SQL-92, SQL-99, and SQL:2008, and each system has its own set of data types, comparison operators, string manipulation features, and so on.

Semantic Heterogeneity. Semantic heterogeneity occurs when there are differences in the meaning, interpretation, and intended use of the same or related data. Semantic heterogeneity among component database systems (DBSs) creates the biggest hurdle in designing global schemas of heterogeneous databases. The **design autonomy** of component DBSs refers to their freedom of choosing the following design parameters, which in turn affect the eventual complexity of the FDBS:

- **The universe of discourse from which the data is drawn.** For example, for two customer accounts, databases in the federation may be from the United States and Japan and have entirely different sets of attributes about customer accounts required by the accounting practices. Currency rate fluctuations would also present a problem. Hence, relations in these two databases that have identical names—CUSTOMER or ACCOUNT—may have some common and some entirely distinct information.
- **Representation and naming.** The representation and naming of data elements and the structure of the data model may be prespecified for each local database.
- **The understanding, meaning, and subjective interpretation of data.** This is a chief contributor to semantic heterogeneity.
- **Transaction and policy constraints.** These deal with serializability criteria, compensating transactions, and other transaction policies.
- **Derivation of summaries.** Aggregation, summarization, and other data-processing features and operations supported by the system.

The above problems related to semantic heterogeneity are being faced by all major multinational and governmental organizations in all application areas. In today's commercial environment, most enterprises are resorting to heterogeneous FDBSs, having heavily invested in the development of individual database systems using diverse data models on different platforms over the last 20 to 30 years. Enterprises are using various forms of software—typically called the **middleware**, or Web-based packages called **application servers** (for example, WebLogic or WebSphere) and even generic systems, called **Enterprise Resource Planning (ERP) systems** (for example, SAP, J. D. Edwards ERP)—to manage the transport of queries and transactions from the global application to individual databases (with possible additional processing for business rules) and the data from the heterogeneous database servers to the global application. Detailed discussion of these types of software systems is outside the scope of this book.

Just as providing the ultimate transparency is the goal of any distributed database architecture, local component databases strive to preserve autonomy. **Communication autonomy** of a component DBS refers to its ability to decide whether to communicate with another component DBS. **Execution autonomy**

refers to the ability of a component DBS to execute local operations without interference from external operations by other component DBSs and its ability to decide the order in which to execute them. The **association autonomy** of a component DBS implies that it has the ability to decide whether and how much to share its functionality (operations it supports) and resources (data it manages) with other component DBSs. The major challenge of designing FDBSs is to let component DBSs interoperate while still providing the above types of autonomies to them.

25.3 Distributed Database Architectures

In this section, we first briefly point out the distinction between parallel and distributed database architectures. While both are prevalent in industry today, there are various manifestations of the distributed architectures that are continuously evolving among large enterprises. The parallel architecture is more common in high-performance computing, where there is a need for multiprocessor architectures to cope with the volume of data undergoing transaction processing and warehousing applications. We then introduce a generic architecture of a distributed database. This is followed by discussions on the architecture of three-tier client-server and federated database systems.

25.3.1 Parallel versus Distributed Architectures

There are two main types of multiprocessor system architectures that are commonplace:

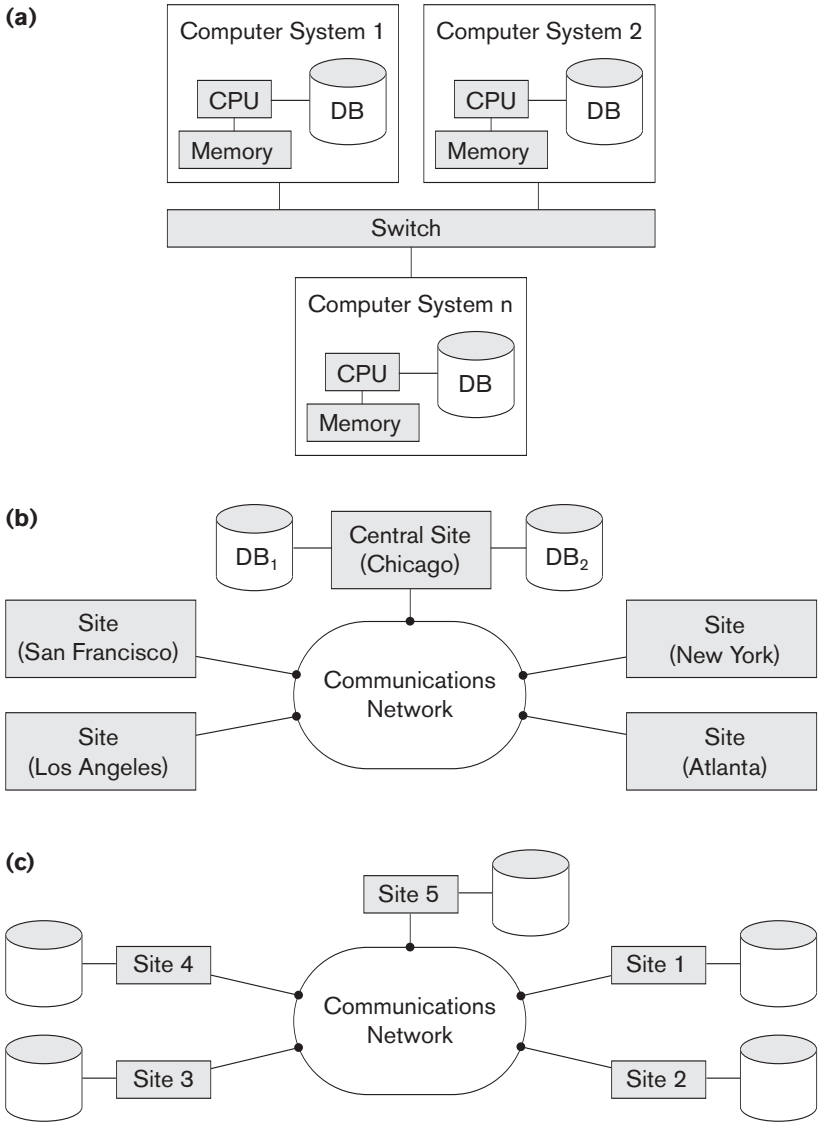
- **Shared memory (tightly coupled) architecture.** Multiple processors share secondary (disk) storage and also share primary memory.
- **Shared disk (loosely coupled) architecture.** Multiple processors share secondary (disk) storage but each has their own primary memory.

These architectures enable processors to communicate without the overhead of exchanging messages over a network.[4] Database management systems developed using the above types of architectures are termed **parallel database management systems** rather than DDBMSs, since they utilize parallel processor technology. Another type of multiprocessor architecture is called **shared nothing architecture**. In this architecture, every processor has its own primary and secondary (disk) memory, no common memory exists, and the processors communicate over a high-speed interconnection network (bus or switch). Although the shared nothing architecture resembles a distributed database computing environment, major differences exist in the mode of operation. In shared nothing multiprocessor systems, there is symmetry and homogeneity of nodes; this is not true of the distributed database environment where heterogeneity of hardware and operating system at each node is very common. Shared nothing architecture is also considered as an environment for

[4]If both primary and secondary memories are shared, the architecture is also known as *shared everything architecture*.

parallel databases. Figure 25.3a illustrates a parallel database (shared nothing), whereas Figure 25.3b illustrates a centralized database with distributed access and Figure 25.3c shows a pure distributed database. We will not expand on parallel architectures and related data management issues here.

Figure 25.3
Some different database system architectures. (a) Shared nothing architecture. (b) A networked architecture with a centralized database at one of the sites. (c) A truly distributed database architecture.



25.3.2 General Architecture of Pure Distributed Databases

In this section we discuss both the logical and component architectural models of a DDB. In Figure 25.4, which describes the generic schema architecture of a DDB, the enterprise is presented with a consistent, unified view showing the logical structure of underlying data across all nodes. This view is represented by the global conceptual schema (GCS), which provides network transparency (see Section 25.1.2). To accommodate potential heterogeneity in the DDB, each node is shown as having its own local internal schema (LIS) based on physical organization details at that particular site. The logical organization of data at each site is specified by the local conceptual schema (LCS). The GCS, LCS, and their underlying mappings provide the fragmentation and replication transparency discussed in Section 25.1.2. Figure 25.5 shows the component architecture of a DDB. It is an extension of its centralized counterpart (Figure 2.3) in Chapter 2. For the sake of simplicity, common elements are not shown here. The global query compiler references the global conceptual schema from the global system catalog to verify and impose defined constraints. The global query optimizer references both global and local conceptual schemas and generates optimized local queries from global queries. It evaluates all candidate strategies using a cost function that estimates cost based on response time (CPU,

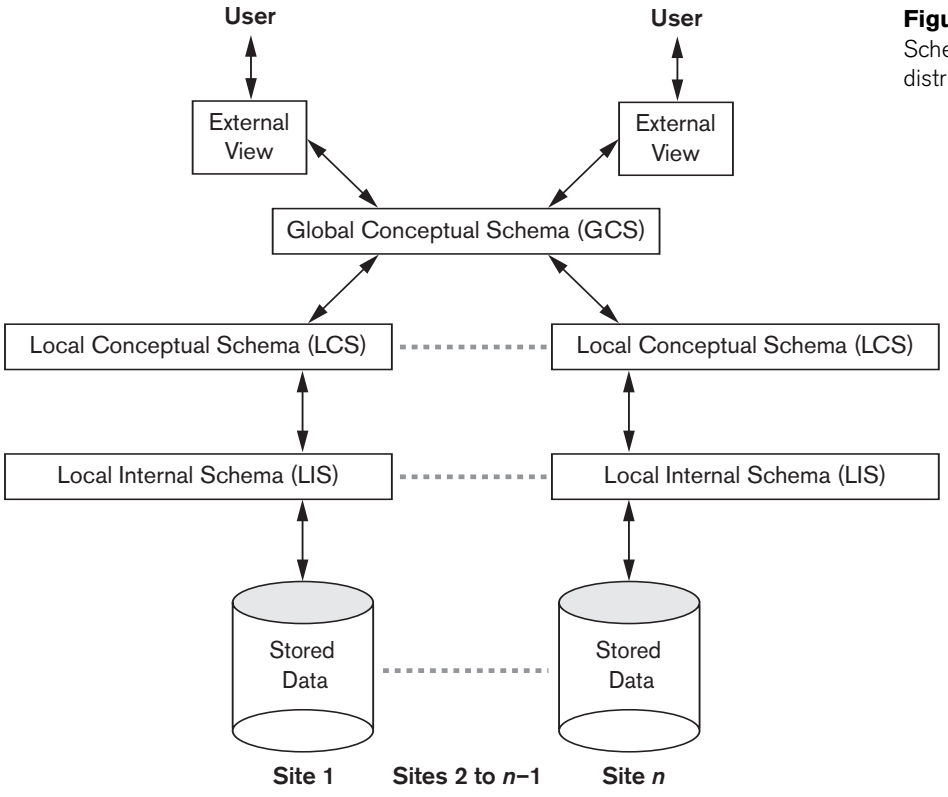


Figure 25.4
Schema architecture of distributed databases.

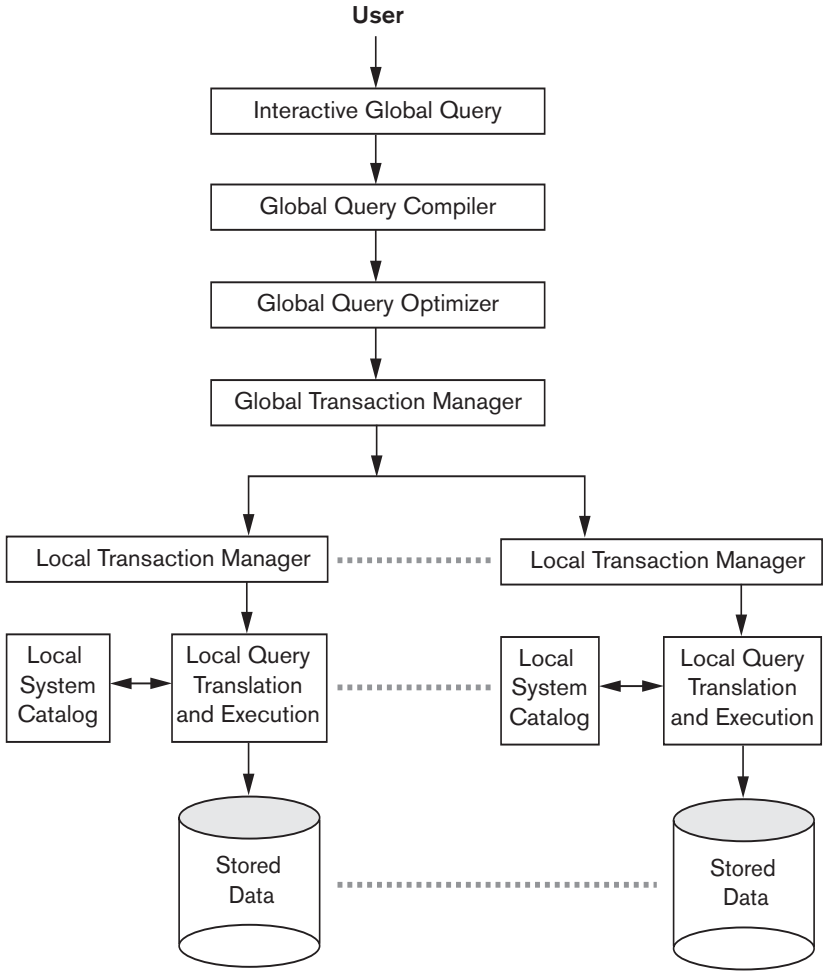


Figure 25.5
Component architecture of distributed databases.

I/O, and network latencies) and estimated sizes of intermediate results. The latter is particularly important in queries involving joins. Having computed the cost for each candidate, the optimizer selects the candidate with the minimum cost for execution. Each local DBMS would have their local query optimizer, transaction manager, and execution engines as well as the local system catalog, which houses the local schemas. The global transaction manager is responsible for coordinating the execution across multiple sites in conjunction with the local transaction manager at those sites.

25.3.3 Federated Database Schema Architecture

Typical five-level schema architecture to support global applications in the FDBS environment is shown in Figure 25.6. In this architecture, the **local schema** is the

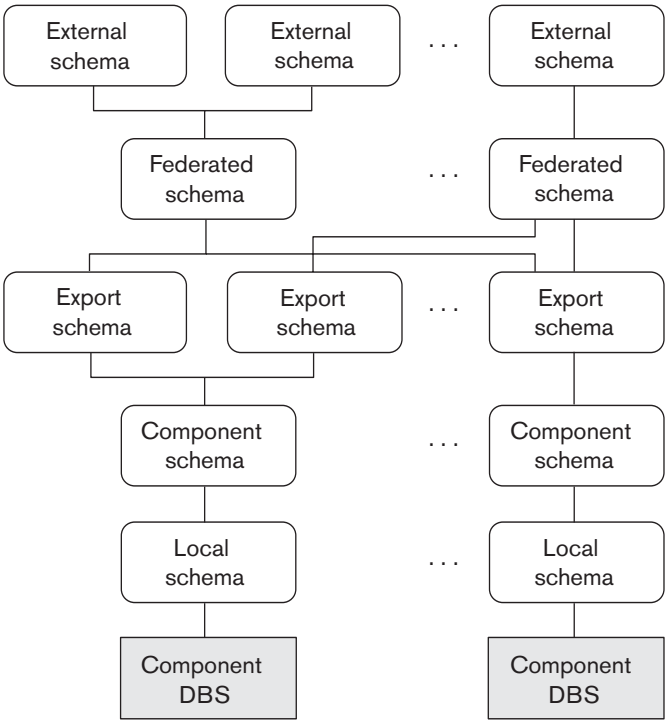


Figure 25.6
The five-level schema architecture in a federated database system (FDBS).

Source: Adapted from Sheth and Larson, "Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases." *ACM Computing Surveys* (Vol. 22: No. 3, September 1990).

conceptual schema (full database definition) of a component database, and the **component schema** is derived by translating the local schema into a canonical data model or common data model (CDM) for the FDBS. Schema translation from the local schema to the component schema is accompanied by generating mappings to transform commands on a component schema into commands on the corresponding local schema. The **export schema** represents the subset of a component schema that is available to the FDBS. The **federated schema** is the global schema or view, which is the result of integrating all the shareable export schemas. The **external schemas** define the schema for a user group or an application, as in the three-level schema architecture.[5]

All the problems related to query processing, transaction processing, and directory and metadata management and recovery apply to FDBSs with additional considerations. It is not within our scope to discuss them in detail here.

[5]For a detailed discussion of the autonomies and the five-level architecture of FDBMSs, see Sheth and Larson (1990).

25.3.4 An Overview of Three-Tier Client-Server Architecture

As we pointed out in the chapter introduction, full-scale DDBMSs have not been developed to support all the types of functionalities that we have discussed so far. Instead, distributed database applications are being developed in the context of the client-server architectures. We introduced the two-tier client-server architecture in Section 2.5. It is now more common to use a three-tier architecture, particularly in Web applications. This architecture is illustrated in Figure 25.7.

In the three-tier client-server architecture, the following three layers exist:

1. **Presentation layer (client).** This provides the user interface and interacts with the user. The programs at this layer present Web interfaces or forms to the client in order to interface with the application. Web browsers are often utilized, and the languages and specifications used include HTML, XHTML, CSS, Flash, MathML, Scalable Vector Graphics (SVG), Java, JavaScript, Adobe Flex, and others. This layer handles user input, output, and navigation by accepting user commands and displaying the needed information, usually in the form of static or dynamic Web pages. The latter are employed when the interaction involves database access. When a Web interface is used, this layer typically communicates with the application layer via the HTTP protocol.
2. **Application layer (business logic).** This layer programs the application logic. For example, queries can be formulated based on user input from the client, or query results can be formatted and sent to the client for presentation. Additional application functionality can be handled at this layer, such

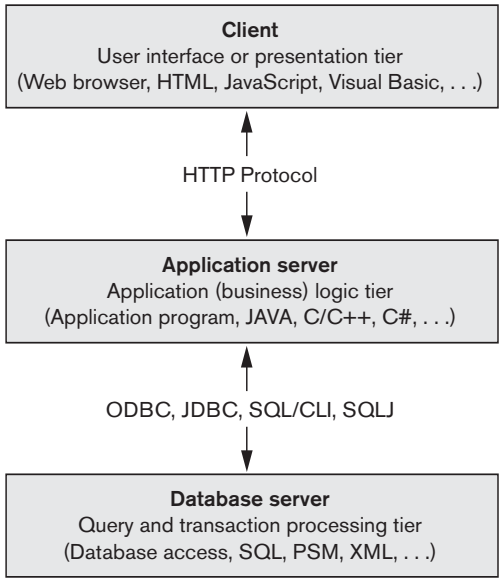


Figure 25.7
The three-tier client-server architecture.

as security checks, identity verification, and other functions. The application layer can interact with one or more databases or data sources as needed by connecting to the database using ODBC, JDBC, SQL/CLI, or other database access techniques.

3. **Database server.** This layer handles query and update requests from the application layer, processes the requests, and sends the results. Usually SQL is used to access the database if it is relational or object-relational and stored database procedures may also be invoked. Query results (and queries) may be formatted into XML (see Chapter 12) when transmitted between the application server and the database server.

Exactly how to divide the DBMS functionality between the client, application server, and database server may vary. The common approach is to include the functionality of a centralized DBMS at the database server level. A number of relational DBMS products have taken this approach, where an **SQL server** is provided. The application server must then formulate the appropriate SQL queries and connect to the database server when needed. The client provides the processing for user interface interactions. Since SQL is a relational standard, various SQL servers, possibly provided by different vendors, can accept SQL commands through standards such as ODBC, JDBC, and SQL/CLI (see Chapter 13).

In this architecture, the application server 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. Interaction between an application server and database server might proceed as follows during the processing of an SQL query:

1. The application server formulates a user query based on input from the client layer and decomposes it into a number of independent site queries. Each site query is sent to the appropriate database server site.
2. Each database server processes the local query and sends the results to the application server site. Increasingly, XML is being touted as the standard for data exchange (see Chapter 12), so the database server may format the query result into XML before sending it to the application server.
3. The application server combines the results of the subqueries to produce the result of the originally required query, formats it into HTML or some other form accepted by the client, and sends it to the client site for display.

The application server 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. Another function controlled by the application server (or coordinator) is that of ensuring consistency of replicated copies of a data item by employing distributed (or global) concurrency control techniques. The application server must also ensure the atomicity of global transactions by performing global recovery when certain sites fail.

If the DDBMS has the capability to *hide* the details of data distribution from the application server, then it enables the application server to execute 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 applications are aware of the details of data distribution.

25.4 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**, which 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 directory** that is accessed by the DDBS applications as needed.

25.4.1 Data Fragmentation

In a DDB, 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 relation—or portion of a relation—is stored at one site only. We discuss replication and its effects later in this section. We also use the terminology of relational databases, but 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 3.5.

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 in Figure 3.5. In many cases, however, a relation can be divided into smaller logical units for distribution. For example, consider the company database shown in Figure 3.6, and assume there are three computer sites—one for each department in the company.[6]

We may want to store the database information relating to each department at the computer site for that department. A technique called *horizontal fragmentation* can be used to partition each relation by department.

[6]Of course, in an actual situation, there will be many more tuples in the relation than those shown in Figure 3.6.

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 in Figure 3.6 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, 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. **Derived horizontal fragmentation** applies the partitioning of a primary relation (DEPARTMENT in our example) to other secondary relations (EMPLOYEE and PROJECT in our example), which are related to the primary via a foreign key. This way, related data between the primary and the secondary relations gets fragmented in the same way.

Vertical Fragmentation. Each site may not need all the attributes of a relation, which would indicate the need for a different type of fragmentation. **Vertical fragmentation** divides a relation "vertically" by columns. 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, Super_ssn, and 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 or some candidate 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.

Notice that each horizontal fragment on a relation R can be specified in the relational algebra by a $\sigma_{C_i}(R)$ operation. A set of horizontal fragments whose conditions $C_1, C_2, ..., C_n$ include all the tuples in R—that is, every tuple in R satisfies (C_1 OR C_2 OR ... 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 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 $L_1, L_2, ..., L_n$ 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:

- $L_1 \cup L_2 \cup ... \cup L_n = \text{ATTRS}(R)$.
- $L_i \cap L_j = \text{PK}(R)$ 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 vertical fragments (assuming no horizontal fragmentation is used). Notice that we could also apply a FULL OUTER JOIN operation and get the same result for a complete vertical fragmentation, even when some horizontal fragmentation may also have been applied. The two vertical fragments of the EMPLOYEE relation with projection lists L_1 = {Ssn, Name, Bdate, Address, Sex} and L_2 = {Ssn, Salary, Super_ssn, Dno} constitute a complete vertical fragmentation of EMPLOYEE.

Two horizontal fragments that are neither complete nor disjoint are those defined on the EMPLOYEE relation in Figure 3.5 by the conditions (Salary > 50000) and (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 = {Name, Address} and L_2 = {Ssn, Name, Salary}; these lists violate both conditions of a complete vertical fragmentation.

Mixed (Hybrid) 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 $C = \text{TRUE}$ (that is, all tuples are selected) 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.

25.4.2 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 results of such queries 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 was no replication, as we will see in Section 25.7.

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 may not. The number of copies of each fragment can range from one up to the total number of sites in the distributed system. A special case of partial replication is occurring heavily in applications where mobile workers—such as sales forces, financial planners, and claims adjustors—carry partially replicated databases with them on laptops and PDAs and synchronize them periodically with the server database.[7] 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, transactions can be submitted at any site, and 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.

[7]For a proposed scalable approach to synchronize partially replicated databases, see Mahajan et al. (1998).

25.4.3 Example of Fragmentation, Allocation, and Replication

We now consider an example of fragmenting and distributing the company database in Figures 3.5 and 3.6. 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 Super_ssn 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 in Figure 3.6 can be stored at site 1. To determine the fragments to be replicated at sites 2 and 3, first we can horizontally fragment DEPARTMENT by its key Dnumber. Then we apply derived fragmentation to the EMPLOYEE, PROJECT, and DEPT_LOCATIONS relations based on their foreign keys for department number—called Dno, Dnum, and Dnumber, respectively, in Figure 3.5. We can vertically fragment the resulting EMPLOYEE fragments to include only the attributes {Name, Ssn, Salary, Super_ssn, Dno}. Figure 25.8 shows the mixed fragments EMPD_5 and EMPD_4, 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 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 in Figure 3.6. 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 the department in which the employee works (which is expressed by the condition *C*) and then fragment further based on the department that controls the projects that employee is working on, as shown in Figure 25.9.

In Figure 25.9, the union of fragments G_1, G_2, and G_3 gives all WORKS_ON tuples for employees who work for department 5. Similarly, the union of fragments G_4, G_5, and G_6 gives all WORKS_ON tuples for employees who work for department 4. On the other hand, the union of fragments G_1, G_4, and G_7 gives all WORKS_ON tuples for projects controlled by department 5. The condition for each of the fragments G_1 through G_9 is shown in Figure 25.9 The relations that represent M:N relationships, such as WORKS_ON, often have several possible logical fragmentations. In our distribution in Figure 25.8, we choose to include all fragments that can be joined to

Figure 25.8
Allocation of fragments to sites. (a) Relation fragments at site 2 corresponding to department 5. (b) Relation fragments at site 3 corresponding to department 4.

(a)

EMPD_5

Fname	Minit	Lname	Ssn	Salary	Super_ssn	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

DEP_5

Dname	Dnumber	Mgr_ssn	Mgr_start_date
Research	5	333445555	1988-05-22

DEP_5_LOCS

Dnumber	Location
5	Bellaire
5	Sugarland
5	Houston

WORKS_ON_5

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

PROJS_5

Pname	Pnumber	Plocation	Dnum
Product X	1	Bellaire	5
Product Y	2	Sugarland	5
Product Z	3	Houston	5

Data at site 2

(b)

EMPD_4

Fname	Minit	Lname	Ssn	Salary	Super_ssn	Dno
Alicia	J	Zelaya	999887777	25000	987654321	4
Jennifer	S	Wallace	987654321	43000	888665555	4
Ahmad	V	Jabbar	987987987	25000	987654321	4

DEP_4

Dname	Dnumber	Mgr_ssn	Mgr_start_date
Administration	4	987654321	1995-01-01

DEP_4_LOCS

Dnumber	Location
4	Stafford

WORKS_ON_4

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

PROJS_4

Pname	Pnumber	Plocation	Dnum
Computerization	10	Stafford	4
New_benefits	30	Stafford	4

Data at site 3

Figure 25.9
Complete and disjoint fragments of the WORKS_ON relation. (a) Fragments of WORKS_ON for employees working in department 5 (C=[Essn in (SELECT Ssn FROM EMPLOYEE WHERE Dno=5)]). (b) Fragments of WORKS_ON for employees working in department 4 (C=[Essn in (SELECT Ssn FROM EMPLOYEE WHERE Dno=4)]). (c) Fragments of WORKS_ON for employees working in department 1 (C=[Essn in (SELECT Ssn FROM EMPLOYEE WHERE Dno=1)]).

(a) Employees in Department 5

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) Employees in Department 4

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) Employees in Department 1

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))

either an EMPLOYEE tuple or a PROJECT tuple at sites 2 and 3. Hence, we place the union of fragments G_1, G_2, G_3, G_4, and G_7 at site 2 and the union of fragments G_4, G_5, G_6, G_2, and G_8 at site 3. Notice that fragments G_2 and G_4 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 Selected Bibliography at the end of this chapter discusses some of the work done in this area.

25.5 Query Processing and Optimization in Distributed Databases

Now we give an overview of how a DDBMS processes and optimizes a query. First we discuss the steps involved in query processing and then elaborate on the communication costs of processing a distributed query. Finally we discuss a special operation, called a *semijoin,* which is used to optimize some types of queries in a DDBMS. A detailed discussion about optimization algorithms is beyond the scope of this book. We attempt to illustrate optimization principles using suitable examples.[8]

25.5.1 Distributed Query Processing

A distributed database query is processed in stages as follows:

1. **Query Mapping.** The input query on distributed data is specified formally using a query language. It is then translated into an algebraic query on global relations. This translation is done by referring to the global conceptual schema and does not take into account the actual distribution and replication of data. Hence, this translation is largely identical to the one performed in a centralized DBMS. It is first normalized, analyzed for semantic errors, simplified, and finally restructured into an algebraic query.
2. **Localization.** In a distributed database, fragmentation results in relations being stored in separate sites, with some fragments possibly being replicated. This stage maps the distributed query on the global schema to separate queries on individual fragments using data distribution and replication information.
3. **Global Query Optimization**. Optimization consists of selecting a strategy from a list of candidates that is closest to optimal. A list of candidate queries can be obtained by permuting the ordering of operations within a fragment query generated by the previous stage. Time is the preferred unit for measuring cost. The total cost is a weighted combination of costs such as CPU cost, I/O costs, and communication costs. Since DDBs are connected by a network, often the communication costs over the network are the most significant. This is especially true when the sites are connected through a wide area network (WAN).

[8]For a detailed discussion of optimization algorithms, see Ozsu and Valduriez (1999).

4. **Local Query Optimization**. This stage is common to all sites in the DDB. The techniques are similar to those used in centralized systems.

The first three stages discussed above are performed at a central control site, while the last stage is performed locally.

25.5.2 Data Transfer Costs of Distributed Query Processing

We discussed the issues involved in processing and optimizing a query in a centralized DBMS in Chapter 19. 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 sample queries. Suppose that the EMPLOYEE and DEPARTMENT relations in Figure 3.5 are distributed at two sites as shown in Figure 25.10. We will assume in this example that neither relation is fragmented. According to Figure 25.10, 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.*

Figure 25.10
Example to illustrate volume of data transferred.

Site 1:

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno

10,000 records
each record is 100 bytes long
Ssn field is 9 bytes long
Dno field is 4 bytes long
Fname field is 15 bytes long
Lname field is 15 bytes long

Site 2:

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date

100 records
each record is 35 bytes long
Dnumber field is 4 bytes long
Mgr_ssn field is 9 bytes long
Dname field is 10 bytes long

The query is submitted at a distinct site 3, which is called the **result site** because the 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, a total of 1,000,000 + 3,500 = 1,003,500 bytes must be transferred.
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 400,000 + 1,000,000 = 1,400,000 bytes must be transferred.
3. Transfer the DEPARTMENT relation to site 1, execute the join at site 1, and send the result to site 3. In this case, 400,000 + 3,500 = 403,500 bytes must be transferred.

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:

$$Q': \pi_{Fname,Lname,Dname}(\text{DEPARTMENT} \bowtie_{Mgr\_ssn=Ssn} \text{EMPLOYEE})$$

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, a total of 1,000,000 + 3,500 = 1,003,500 bytes must be transferred.
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 = 4,000 bytes, so 4,000 + 1,000,000 = 1,004,000 bytes must be transferred.
3. Transfer the DEPARTMENT relation to site 1, execute the join at site 1, and send the result to site 3. In this case, 4,000 + 3,500 = 7,500 bytes must be transferred.

Again, we would choose strategy 3—this time 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:

1. Transfer the EMPLOYEE relation to site 2, execute the query, and present the result to the user at site 2. Here, the same number of bytes—1,000,000—must be transferred for both Q and Q′.
2. Transfer the DEPARTMENT relation to site 1, execute the query at site 1, and send the result back to site 2. In this case 400,000 + 3,500 = 403,500 bytes must be transferred for Q and 4,000 + 3,500 = 7,500 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.

25.5.3 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 or attributes 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 $F' = \pi_{\text{Mgr\_ssn}}(\text{DEPARTMENT})$, 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 $R = \pi_{\text{Dno, Fname, Lname}}(F \bowtie_{\text{Dnumber=Dno}} \text{EMPLOYEE})$, whose size is $34 * 10{,}000 = 340{,}000$ bytes, whereas, for Q′, we transfer $R' = \pi_{\text{Mgr\_ssn, Fname, Lname}}(F' \bowtie_{\text{Mgr\_ssn=Ssn}} \text{EMPLOYEE})$, whose size is $39 * 100 = 3{,}900$ 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 4,800 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** $R \ltimes_{A=B} S$, where A and B are domain-compatible attributes of R and S, respectively, produces the same result as the relational algebra expression $\pi_R(R \bowtie_{A=B} S)$. In a distributed environment where R and S reside at different sites, the semijoin is typically implemented by first transferring $F = \pi_B(S)$ 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,

$$R \ltimes S \neq S \ltimes R$$

25.5.4 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 25.8, and those at site 1 are shown in Figure 3.6, as in our earlier example. A user who submits such a query must specify whether it references the PROJS_5 and WORKS_ON_5 relations at site 2 (Figure 25.8) or the PROJECT and WORKS_ON relations at site 1 (Figure 3.6). 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 in Figure 3.5 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 to be discussed in Section 25.7. For queries, a **query decomposition** module must break up or **decompose** a query into **subqueries** that can be executed at the individual sites. Additionally, 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 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 3.6) are TRUE (all tuples), and the attribute lists are * (all attributes). For the fragments shown in Figure 25.8, we have the guard conditions and attribute lists shown in Figure 25.11. 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 insert 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

(a) EMPD5
attribute list: Fname, Minit, Lname, Ssn, Salary, Super_ssn, Dno
guard condition: Dno=5
DEP5
attribute list: * (all attributes Dname, Dnumber, Mgr_ssn, Mgr_start_date)
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, Super_ssn, Dno
guard condition: Dno=4
DEP4
attribute list: * (all attributes Dname, Dnumber, Mgr_ssn, Mgr_start_date)
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 25.11
Guard conditions and attributes lists for fragments.
(a) Site 2 fragments. (b) Site 3 fragments.

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 in Figure 3.5 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:

$$T_1 \leftarrow \pi_{\text{Essn}}(\text{PROJS5} \bowtie_{\text{Pnumber=Pno}} \text{WORKS\_ON5})$$
$$T_2 \leftarrow \pi_{\text{Essn, Fname, Lname}}(T_1 \bowtie_{\text{Essn=Ssn}} \text{EMPLOYEE})$$
$$\text{RESULT} \leftarrow \pi_{\text{Fname, Lname, Hours}}(T_2 * \text{WORKS\_ON5})$$

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 T_1 can be executed at site 2, and the projected column Essn can be sent to site 1. Subquery T_2 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.

25.6 Overview of Transaction Management in Distributed Databases

The global and local transaction management software modules, along with the concurrency control and recovery manager of a DDBMS, collectively guarantee the ACID properties of transactions (see Chapter 21). We discuss distributed transaction management in this section and explore concurrency control in Section 25.7.

As can be seen in Figure 25.5, an additional component called the **global transaction manager** is introduced for supporting distributed transactions. The site where the transaction originated can temporarily assume the role of global transaction manager and coordinate the execution of database operations with transaction managers across multiple sites. Transaction managers export their functionality as an interface to the application programs. The operations exported by this interface are similar to those covered in Section 21.2.1, namely BEGIN_TRANSACTION, READ or WRITE, END_TRANSACTION, COMMIT_TRANSACTION, and ROLLBACK (or ABORT). The manager stores bookkeeping information related to each transaction, such as a unique identifier, originating site, name, and so on. For READ operations, it returns a local copy if valid and available. For WRITE operations, it ensures that updates are visible across all sites containing copies (replicas) of the data item. For ABORT operations, the manager ensures that no effects of the transaction are reflected in any site of the distributed database. For COMMIT operations, it ensures that the effects of a write are persistently recorded on all databases containing copies of the data item. Atomic termination (COMMIT/ ABORT) of distributed transactions is commonly implemented using the two-phase commit protocol. We give more details of this protocol in the following section.

The transaction manager passes to the concurrency controller the database operation and associated information. The controller is responsible for acquisition and release of associated locks. If the transaction requires access to a locked resource, it is delayed until the lock is acquired. Once the lock is acquired, the operation is sent to the runtime processor, which handles the actual execution of the database operation. Once the operation is completed, locks are released and the transaction manager is updated with the result of the operation. We discuss commonly used distributed concurrency methods in Section 25.7.

25.6.1 Two-Phase Commit Protocol

In Section 23.6, we described the *two-phase commit protocol* (**2PC**), which requires a **global recovery manager**, or **coordinator**, to maintain information needed for recovery, in addition to the local recovery managers and the information they maintain (log, tables) The two-phase commit protocol has certain drawbacks that led to the development of the three-phase commit protocol, which we discuss next.

25.6.2 Three-Phase Commit Protocol

The biggest drawback of 2PC is that it is a blocking protocol. Failure of the coordinator blocks all participating sites, causing them to wait until the coordinator recovers. This can cause performance degradation, especially if participants are holding locks to shared resources. Another problematic scenario is when both the coordinator and a participant that has committed crash together. In the two-phase commit protocol, a participant has no way to ensure that all participants got the commit message in the second phase. Hence once a decision to commit has been made by the coordinator in the first phase, participants will commit their transactions in the second phase independent of receipt of a global commit message by other participants. Thus, in the situation that both the coordinator and a committed participant crash together, the result of the transaction becomes uncertain or nondeterministic. Since the transaction has already been committed by one participant, it cannot be aborted on recovery by the coordinator. Also, the transaction cannot be optimistically committed on recovery since the original vote of the coordinator may have been to abort.

These problems are solved by the three-phase commit (3PC) protocol, which essentially divides the second commit phase into two subphases called **prepare-to-commit** and **commit**. The prepare-to-commit phase is used to communicate the result of the vote phase to all participants. If all participants vote yes, then the coordinator instructs them to move into the prepare-to-commit state. The commit subphase is identical to its two-phase counterpart. Now, if the coordinator crashes during this subphase, another participant can see the transaction through to completion. It can simply ask a crashed participant if it received a prepare-to-commit message. If it did not, then it safely assumes to abort. Thus the state of the protocol can be recovered irrespective of which participant crashes. Also, by limiting the time required for a transaction to commit or abort to a maximum time-out period, the protocol ensures that a transaction attempting to commit via 3PC releases locks on time-out.

The main idea is to limit the wait time for participants who have committed and are waiting for a global commit or abort from the coordinator. When a participant receives a precommit message, it knows that the rest of the participants have voted to commit. If a precommit message has not been received, then the participant will abort and release all locks.

25.6.3 Operating System Support for Transaction Management

The following are the main benefits of operating system (OS)-supported transaction management:

- Typically, DBMSs use their own semaphores[9] to guarantee mutually exclusive access to shared resources. Since these semaphores are implemented in userspace at the level of the DBMS application software, the OS has no knowledge about them. Hence if the OS deactivates a DBMS process holding a lock, other DBMS processes wanting this lock resource get queued. Such a situation can cause serious performance degradation. OS-level knowledge of semaphores can help eliminate such situations.
- Specialized hardware support for locking can be exploited to reduce associated costs. This can be of great importance, since locking is one of the most common DBMS operations.
- Providing a set of common transaction support operations though the kernel allows application developers to focus on adding new features to their products as opposed to reimplementing the common functionality for each application. For example, if different DDBMSs are to coexist on the same machine and they chose the two-phase commit protocol, then it is more beneficial to have this protocol implemented as part of the kernel so that the DDBMS developers can focus more on adding new features to their products.

25.7 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 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.

[9]Semaphores are data structures used for synchronized and exclusive access to shared resources for preventing race conditions in a parallel computing system.

- **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 the 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 23.6) 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.

25.7.1 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 there. 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 thus is not overly complex. However, it has certain inherent disadvantages. One is that all locking requests are sent to a single site, possibly overloading that site and causing a system bottleneck. A second 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 primary site failure, the backup site takes over as the primary site, and a new backup site is 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 in a tedious recovery process. 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 Selected Bibliography at the end of this chapter discuss the process in detail.

25.7.2 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 includes 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.

25.7.3 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 is often used to ensure the correctness of distributed commit (see Section 23.6).

25.8 Distributed Catalog Management

Efficient catalog management in distributed databases is critical to ensure satisfactory performance related to site autonomy, view management, and data distribution and replication. Catalogs are databases themselves containing metadata about the distributed database system.

Three popular management schemes for distributed catalogs are *centralized* catalogs, *fully replicated* catalogs, and *partitioned* catalogs. The choice of the scheme depends on the database itself as well as the access patterns of the applications to the underlying data.

Centralized Catalogs. In this scheme, the entire catalog is stored in one single site. Owing to its central nature, it is easy to implement. On the other hand, the advantages of reliability, availability, autonomy, and distribution of processing load are adversely impacted. For read operations from noncentral sites, the requested catalog data is locked at the central site and is then sent to the requesting site. On completion of the read operation, an acknowledgement is sent to the central site, which in turn unlocks this data. All update operations must be processed through the central site. This can quickly become a performance bottleneck for write-intensive applications.

Fully Replicated Catalogs. In this scheme, identical copies of the complete catalog are present at each site. This scheme facilitates faster reads by allowing them to be answered locally. However, all updates must be broadcast to all sites. Updates are treated as transactions and a centralized two-phase commit scheme is employed to ensure catalog consitency. As with the centralized scheme, write-intensive applications may cause increased network traffic due to the broadcast associated with the writes.

Partially Replicated Catalogs. The centralized and fully replicated schemes restrict site autonomy since they must ensure a consistent global view of the catalog. Under the partially replicated scheme, each site maintains complete catalog information on data stored locally at that site. Each site is also permitted to cache entries retrieved from remote sites. However, there are no guarantees that these cached copies will be the most recent and updated. The system tracks catalog entries for sites where the object was created and for sites that contain copies of this object. Any changes to copies are propagated immediately to the original (birth) site. Retrieving updated copies to replace stale data may be delayed until an access to this data occurs. In general, fragments of relations across sites should be uniquely accessible. Also, to ensure data distribution transparency, users should be allowed to create synonyms for remote objects and use these synonyms for subsequent referrals.

25.9 Current Trends in Distributed Databases

Current trends in distributed data management are centered on the Internet, in which petabytes of data can be managed in a scalable, dynamic, and reliable fashion. Two important areas in this direction are cloud computing and peer-to-peer databases.

25.9.1 Cloud Computing

Cloud computing is the paradigm of offering computer infrastructure, platforms, and software as services over the Internet. It offers significant economic advantages by limiting both up-front capital investments toward computer infrastructure as well as total cost of ownership. It has introduced a new challenge of managing petabytes of data in a scalable fashion. Traditional database systems for managing enterprise data proved to be inadequate in handling this challenge, which has resulted in a major architectural revision. The Claremont report[10] by a group of senior database researchers envisions that future research in cloud computing will result in the emergence of new data management architectures and the interplay of structured and unstructured data as well as other developments.

Performance costs associated with partial failures and global synchronization were key performance bottlenecks of traditional database solutions. The key insight is that the hash-value nature of the underlying datasets used by these organizations lends itself naturally to partitioning. For instance, search queries essentially involve a recursive process of mapping keywords to a set of related documents, which can benefit from such a partitioning. Also, the partitions can be treated independently, thereby eliminating the need for a coordinated commit. Another problem with traditional DDBMSs is the lack of support for efficient dynamic partitioning of data, which limited scalability and resource utilization. Traditional systems treated system metadata and application data alike, with the system data requiring strict consistency and availability guarantees. But application data has variable requirements on these characteristics, depending on its nature. For example, while a search engine can afford weaker consistency guarantees, an online text editor like Google Docs, which allows concurrent users, has strict consistency requirements.

The metadata of a distributed database system should be decoupled from its actual data in order to ensure scalability. This decoupling can be used to develop innovative solutions to manage the actual data by exploiting their inherent suitability to partitioning and using traditional database solutions to manage critical system metadata. Since metadata is only a fraction of the total data set, it does not prove to be a performance bottleneck. Single object semantics of these implementations enables higher tolerance to nonavailability of certain sections of data. Access to data is typically by a single object in an atomic fashion. Hence, transaction support to such data is not as stringent as for traditional databases.[11] There is a varied set of

[10]"The Claremont Report on Database Research" is available at http://db.cs.berkeley.edu/claremont/claremontreport08.pdf.

[11]Readers may refer to the work done by Das et al. (2008) for further details.

cloud services available today, including application services (salesforce.com), storage services (Amazon Simple Storage Service, or Amazon S3), compute services (Google App Engine, Amazon Elastic Compute Cloud—Amazon EC2), and data services (Amazon SimpleDB, Microsoft SQL Server Data Services, Google's Datastore). More and more data-centric applications are expected to leverage data services in the cloud. While most current cloud services are data-analysis intensive, it is expected that business logic will eventually be migrated to the cloud. The key challenge in this migration would be to ensure the scalability advantages for multiple object semantics inherent to business logic. For a detailed treatment of cloud computing, refer to the relevant bibliographic references in this chapter's Selected Bibliography.

25.9.2 Peer-to-Peer Database Systems

A peer-to-peer database system (PDBS) aims to integrate advantages of P2P (peer-to-peer) computing, such as scalability, attack resilience, and self-organization, with the features of decentralized data management. Nodes are autonomous and are linked only to a small number of peers individually. It is permissible for a node to behave purely as a collection of files without offering a complete set of traditional DBMS functionality. While FDBS and MDBS mandate the existence of mappings between local and global federated schemas, PDBSs attempt to avoid a global schema by providing mappings between pairs of information sources. In PDBS, each peer potentially models semantically related data in a manner different from other peers, and hence the task of constructing a central mediated schema can be very challenging. PDBSs aim to decentralize data sharing. Each peer has a schema associated with its domain-specific stored data. The PDBS constructs a semantic path[12] of mappings between peer schemas. Using this path, a peer to which a query has been submitted can obtain information from any relevant peer connected through this path. In multidatabase systems, a separate global query processor is used, whereas in a P2P system a query is shipped from one peer to another until it is processed completely. A query submitted to a node may be forwarded to others based on the mapping graph of semantic paths. Edutella and Piazza are examples of PDBSs. Details of these systems can be found from the sources mentioned in this chapter's Selected Bibliography.

25.10 Distributed Databases in Oracle[13]

Oracle provides support for homogeneous, heterogeneous, and client server architectures of distributed databases. In a homogeneous architecture, a minimum of two Oracle databases reside on at least one machine. Although the location and platform of the databases are transparent to client applications, they would need to

[12]A **semantic path** describes the higher-level relationship between two domains that are dissimilar but not unrelated.

[13]The discussion is based on available documentation at http://docs.oracle.com.

distinguish between local and remote objects semantically. Using synonyms, this need can be overcome wherein users can access the remote objects with the same syntax as local objects. Different versions of DBMSs can be used, although it must be noted that Oracle offers backward compatibility but not forward compatibility between its versions. For example, it is possible that some of the SQL extensions that were incorporated into Oracle 11i may not be understood by Oracle 9.

In a heterogeneous architecture, at least one of the databases in the network is a non-Oracle system. The Oracle database local to the application hides the underlying heterogeneity and offers the view of a single local, underlying Oracle database. Connectivity is handled by use of an ODBC- or OLE-DB-compliant protocol or by Oracle's Heterogeneous Services and Transparent Gateway agent components. A discussion of the Heterogeneous Services and Transparent Gateway agents is beyond the scope of this book, and the reader is advised to consult the online Oracle documentation.

In the client-server architecture, the Oracle database system is divided into two parts: a front end as the client portion, and a back end as the server portion. The client portion is the front-end database application that interacts with the user. The client has no data access responsibility and merely handles the requesting, processing, and presentation of data managed by the server. The server portion runs Oracle and handles the functions related to concurrent shared access. It accepts SQL and PL/SQL statements originating from client applications, processes them, and sends the results back to the client. Oracle client-server applications provide location transparency by making the location of data transparent to users; several features like views, synonyms, and procedures contribute to this. Global naming is achieved by using <TABLE_NAME@DATABASE_NAME> to refer to tables uniquely.

Oracle uses a two-phase commit protocol to deal with concurrent distributed transactions. The COMMIT statement triggers the two-phase commit mechanism. The RECO (recoverer) background process automatically resolves the outcome of those distributed transactions in which the commit was interrupted. The RECO of each local Oracle server automatically commits or rolls back any *in-doubt* distributed transactions consistently on all involved nodes. For long-term failures, Oracle allows each local DBA to manually commit or roll back any in-doubt transactions and free up resources. Global consistency can be maintained by restoring the database at each site to a predetermined fixed point in the past.

Oracle's distributed database architecture is shown in Figure 25.12. A node in a distributed database system can act as a client, as a server, or both, depending on the situation. The figure shows two sites where databases called HQ (headquarters) and Sales are kept. For example, in the application shown running at the headquarters, for an SQL statement issued against local data (for example, DELETE FROM DEPT ...), the HQ computer acts as a server, whereas for a statement against remote data (for example, INSERT INTO EMP@SALES), the HQ computer acts as a client.

Communication in such a distributed heterogeneous environment is facilitated through Oracle Net Services, which supports standard network protocols and APIs. Under Oracle's client-server implementation of distributed databases, Net Services

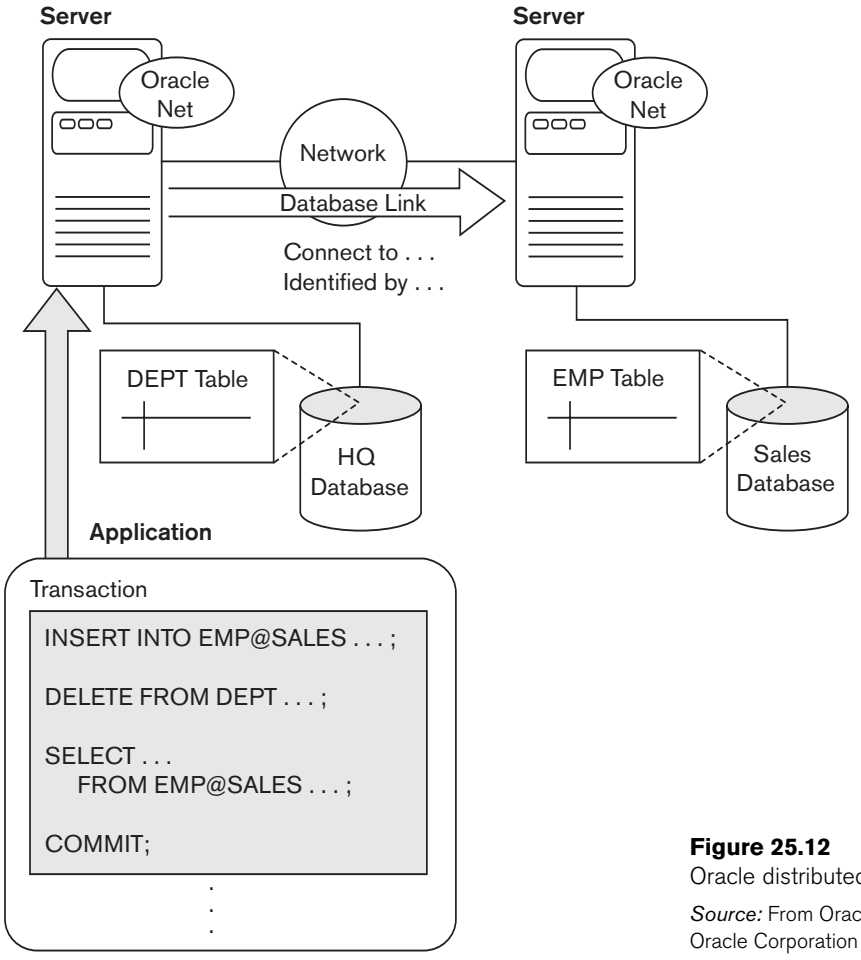


Figure 25.12
Oracle distributed database system.

Source: From Oracle (2008). Copyright © Oracle Corporation 2008. All rights reserved.

is responsible for establishing and managing connections between a client application and database server. It is present in each node on the network running an Oracle client application, database server, or both. It packages SQL statements into one of the many communication protocols to facilitate client-to-server communication and then packages the results back similarly to the client. The support offered by Net Services to heterogeneity refers to platform specifications only and not the database software. Support for DBMSs other than Oracle is through Oracle's Heterogeneous Services and Transparent Gateway. Each database has a unique global name provided by a hierarchical arrangement of network domain names that is prefixed to the database name to make it unique.

Oracle supports database links that define a one-way communication path from one Oracle database to another. For example,

CREATE DATABASE LINK sales.us.americas;

establishes a connection to the sales database in Figure 25.12 under the network domain us that comes under domain americas. Using links, a user can access a remote object on another database subject to ownership rights without the need for being a user on the remote database.

Data in an Oracle DDBS can be replicated using snapshots or replicated master tables. Replication is provided at the following levels:

- **Basic replication.** Replicas of tables are managed for read-only access. For updates, data must be accessed at a single primary site.
- **Advanced (symmetric) replication.** This extends beyond basic replication by allowing applications to update table replicas throughout a replicated DDBS. Data can be read and updated at any site. This requires additional software called *Oracle's advanced replication option*. A **snapshot** generates a copy of a part of the table by means of a query called the *snapshot defining query*. A simple snapshot definition looks like this:

```
CREATE SNAPSHOT SALES_ORDERS AS
SELECT * FROM SALES_ORDERS@hq.us.americas;
```

Oracle groups snapshots into refresh groups. By specifying a refresh interval, the snapshot is automatically refreshed periodically at that interval by up to ten **Snapshot Refresh Processes (SNPs)**. If the defining query of a snapshot contains a distinct or aggregate function, a GROUP BY or CONNECT BY clause, or join or set operations, the snapshot is termed a **complex snapshot** and requires additional processing. Oracle (up to version 7.3) also supports ROWID snapshots that are based on physical row identifiers of rows in the master table.

Heterogeneous Databases in Oracle. In a heterogeneous DDBS, at least one database is a non-Oracle system. **Oracle Open Gateways** provides access to a non-Oracle database from an Oracle server, which uses a database link to access data or to execute remote procedures in the non-Oracle system. The Open Gateways feature includes the following:

- **Distributed transactions.** Under the two-phase commit mechanism, transactions may span Oracle and non-Oracle systems.
- **Transparent SQL access.** SQL statements issued by an application are transparently transformed into SQL statements understood by the non-Oracle system.
- **Pass-through SQL and stored procedures.** An application can directly access a non-Oracle system using that system's version of SQL. Stored procedures in a non-Oracle SQL-based system are treated as if they were PL/SQL remote procedures.
- **Global query optimization.** Cardinality information, indexes, and so on at the non-Oracle system are accounted for by the Oracle server query optimizer to perform global query optimization.
- **Procedural access.** Procedural systems like messaging or queuing systems are accessed by the Oracle server using PL/SQL remote procedure calls.

In addition to the above, data dictionary references are translated to make the non-Oracle data dictionary appear as a part of the Oracle server's dictionary. Character set translations are done between national language character sets to connect multilingual databases.

From a security perspective, Oracle recommends that if a query originates at site A and accesses sites B, C, and D, then the auditing of links should be done in the database at site A only. This is because the remote databases cannot distinguish whether a successful connection request and following SQL statements are coming from another server or a locally connected client.

25.10.1 Directory Services

A concept closely related with distributed enterprise systems is **online directories**. Online directories are essentially a structured organization of metadata needed for management functions. They can represent information about a variety of sources ranging from security credentials, shared network resources, and database catalog. **Lightweight Directory Access Protocol** (LDAP) is an industry standard protocol for directory services. LDAP enables the use of a partitioned **Directory Information Tree** (DIT) across multiple LDAP servers, which in turn can return references to other servers as a result of a directory query. Online directories and LDAP are particularly important in distributed databases, wherein access of metadata related to transparencies discussed in Section 25.1 must be scalable, secure, and highly available.

Oracle supports LDAP Version 3 and online directories through Oracle Internet Directory, a general-purpose directory service for fast access and centralized management of metadata pertaining to distributed network resources and users. It runs as an application on an Oracle database and communicates with the database through Oracle Net Services. It also provides password-based, anonymous, and certificate-based user authentication using SSL Version 3.

Figure 25.13 illustrates the architecture of the Oracle Internet Directory. The main components are:

- **Oracle directory server**. Handles client requests and updates for information pertaining to people and resources.
- **Oracle directory replication server.** Stores a copy of the LDAP data from Oracle directory servers as a backup.
- **Directory administrator:** Supports both GUI-based and command line-based interfaces for directory administration.

25.11 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. First we discussed the reasons for distribution and the potential advantages of distributed databases over centralized systems. Then the concept of

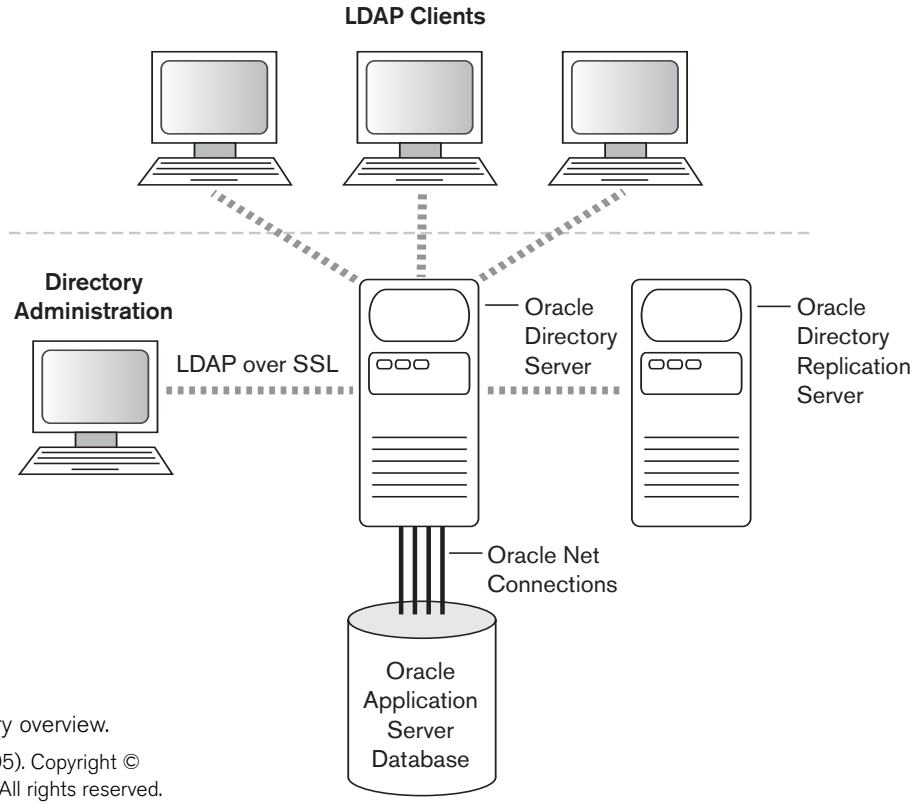


Figure 25.13
Oracle Internet Directory overview.

Source: From Oracle (2005). Copyright © Oracle Corporation 2005. All rights reserved.

distribution transparency and the related concepts of fragmentation transparency and replication transparency were defined. We categorized DDBMSs by using criteria such as the degree of homogeneity of software modules and the degree of local autonomy. We distinguished between parallel and distributed system architectures and then introduced the generic architecture of distributed databases from both a component as well as a schematic architectural perspective. The issues of federated database management were then discussed in some detail, focusing on the needs of supporting various types of autonomies and dealing with semantic heterogeneity. We also reviewed the client-server architecture concepts and related them to distributed databases. We discussed the design issues related to data fragmentation, replication, and distribution, and we distinguished between horizontal and vertical fragments of relations. The use of data replication to improve system reliability and availability was then discussed. We illustrated some of the techniques used in distributed query processing and discussed the cost of communication among sites, which is considered a major factor in distributed query optimization. The different techniques for executing joins were compared and we then presented the semijoin technique for joining relations that reside on different sites. Then we discussed transaction management, including different commit protocols and operating system support for transaction management. We briefly discussed the concurrency

control and recovery techniques used in DDBMSs, and then reviewed some of the additional problems that must be dealt with in a distributed environment that do not appear in a centralized environment. We reviewed catalog management in distributed databases and summarized their relative advantages and disadvantages. We then introduced Cloud Computing and Peer to Peer Database Systems as new focus areas in DDBs in response to the need of managing petabytes of information accessible over the Internet today.

We described some of the facilities in Oracle to support distributed databases. We also discussed online directories and the LDAP protocol in brief.

Review Questions

25.1. What are the main reasons for and potential advantages of distributed databases?

25.2. What additional functions does a DDBMS have over a centralized DBMS?

25.3. 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*.

25.4. Discuss the architecture of a DDBMS. Within the context of a centralized DBMS, briefly explain new components introduced by the distribution of data.

25.5. 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.

25.6. Compare the two-tier and three-tier client-server architectures.

25.7. What is a fragment of a relation? What are the main types of fragments? Why is fragmentation a useful concept in distributed database design?

25.8. Why is data replication useful in DDBMSs? What typical units of data are replicated?

25.9. What is meant by *data allocation* in distributed database design? What typical units of data are distributed over sites?

25.10. How is a horizontal partitioning of a relation specified? How can a relation be put back together from a complete horizontal partitioning?

25.11. How is a vertical partitioning of a relation specified? How can a relation be put back together from a complete vertical partitioning?

25.12. Discuss the naming problem in distributed databases.

25.13. What are the different stages of processing a query in a DDBMS?

25.14. Discuss the different techniques for executing an equijoin of two files located at different sites. What main factors affect the cost of data transfer?

25.15. Discuss the semijoin method for executing an equijoin of two files located at different sites. Under what conditions is an equijoin strategy efficient?

25.16. Discuss the factors that affect query decomposition. How are guard conditions and attribute lists of fragments used during the query decomposition process?

25.17. 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?

25.18. List the support offered by operating systems to a DDBMS and also their benefits.

25.19. Discuss the factors that do not appear in centralized systems that affect concurrency control and recovery in distributed systems.

25.20. Discuss the two-phase commit protocol used for transaction management in a DDBMS. List its limitations and explain how they are overcome using the three-phase commit protocol.

25.21. Compare the primary site method with the primary copy method for distributed concurrency control. How does the use of backup sites affect each?

25.22. When are voting and elections used in distributed databases?

25.23. Discuss catalog management in distributed databases.

25.24. What are the main challenges facing a traditional DDBMS in the context of today's Internet applications? How does cloud computing attempt to address them?

25.25. Discuss briefly the support offered by Oracle for homogeneous, heterogeneous, and client-server based distributed database architectures.

25.26. Discuss briefly online directories, their management, and their role in distributed databases.

Exercises

25.27. Consider the data distribution of the COMPANY database, where the fragments at sites 2 and 3 are as shown in Figure 25.9 and the fragments at site 1 are as shown in Figure 3.6. 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?

a. For each employee in department 5, retrieve the employee name and the names of the employee's dependents.

b. Print the names of all employees who work in department 5 but who work on some project *not* controlled by department 5.

25.28. Consider the following relations:

BOOKS(Book#, Primary_author, Topic, Total_stock, $price)
BOOKSTORE(Store#, City, State, Zip, Inventory_value)
STOCK(Store#, Book#, Qty)

Total_stock is the total number of books in stock and Inventory_value is the total inventory value for the store in dollars.

a. Give an example of two simple predicates that would be meaningful for the BOOKSTORE relation for horizontal partitioning.

b. How would a derived horizontal partitioning of STOCK be defined based on the partitioning of BOOKSTORE?

c. Show predicates by which BOOKS may be horizontally partitioned by topic.

d. Show how the STOCK may be further partitioned from the partitions in (b) by adding the predicates in (c).

25.29. Consider a distributed database for a bookstore chain called National Books with three sites called EAST, MIDDLE, and WEST. The relation schemas are given in Exercise 25.28. Consider that BOOKS are fragmented by $price amounts into:

B_1: BOOK1: $price up to $20
B_2: BOOK2: $price from $20.01 to $50
B_3: BOOK3: $price from $50.01 to $100
B_4: BOOK4: $price $100.01 and above

Similarly, BOOK_STORES are divided by ZIP Codes into:

S_1: EAST: Zip up to 35000
S_2: MIDDLE: Zip 35001 to 70000
S_3: WEST: Zip 70001 to 99999

Assume that STOCK is a derived fragment based on BOOKSTORE only.

a. Consider the query:

```
SELECT   Book#, Total_stock
FROM     Books
WHERE    $price > 15 AND $price < 55;
```

Assume that fragments of BOOKSTORE are nonreplicated and assigned based on region. Assume further that BOOKS are allocated as:

EAST: B_1, B_4
MIDDLE: B_1, B_2
WEST: B_1, B_2, B_3, B_4

Assuming the query was submitted in EAST, what remote subqueries does it generate? (Write in SQL.)

b. If the price of Book#= 1234 is updated from $45 to $55 at site MIDDLE, what updates does that generate? Write in English and then in SQL.

c. Give a sample query issued at WEST that will generate a subquery for MIDDLE.

d. Write a query involving selection and projection on the above relations and show two possible query trees that denote different ways of execution.

25.30. Consider that you have been asked to propose a database architecture in a large organization (General Motors, for example) to consolidate all data including legacy databases (from hierarchical and network models, which are explained in the Web Appendices D and E; no specific knowledge of these models is needed) as well as relational databases, which are geographically distributed so that global applications can be supported. Assume that alternative one is to keep all databases as they are, while alternative two is to first convert them to relational and then support the applications over a distributed integrated database.

a. Draw two schematic diagrams for the above alternatives showing the linkages among appropriate schemas. For alternative one, choose the approach of providing export schemas for each database and constructing unified schemas for each application.

b. List the steps that you would have to go through under each alternative from the present situation until global applications are viable.

c. Compare these from the issues of:
   i. design time considerations
   ii. runtime considerations

Selected Bibliography

The textbooks by Ceri and Pelagatti (1984a) and Ozsu and Valduriez (1999) are devoted to distributed databases. Peterson and Davie (2008), Tannenbaum (2003), and Stallings (2007) cover data communications and computer networks. Comer (2008) discusses networks and internets. Ozsu et al. (1994) has a collection of papers on distributed object management.

Most of the research on distributed database design, query processing, and optimization occurred in the 1980s and 1990s; we quickly review the important references here. Distributed database design has been addressed in terms of horizontal and vertical fragmentation, allocation, and replication. Ceri et al. (1982) defined the concept of minterm horizontal fragments. Ceri et al. (1983) developed an integer programming-based optimization model for horizontal fragmentation and allocation. Navathe et al. (1984) developed algorithms for vertical fragmentation based on attribute affinity and showed a variety of contexts for vertical fragment allocation. Wilson and Navathe (1986) present an analytical model for optimal allocation of fragments. Elmasri et al. (1987) discuss fragmentation for the ECR model; Karlapalem et al. (1996) discuss issues for distributed design of object databases. Navathe et al. (1996) discuss mixed fragmentation by combining horizontal and

vertical fragmentation; Karlapalem et al. (1996) present a model for redesign of distributed databases.

Distributed query processing, optimization, and decomposition are discussed in Hevner and Yao (1979), Kerschberg et al. (1982), Apers et al. (1983), Ceri and Pelagatti (1984), and Bodorick et al. (1992). Bernstein and Goodman (1981) discuss the theory behind semijoin processing. Wong (1983) discusses the use of relationships in relation fragmentation. Concurrency control and recovery schemes are discussed in Bernstein and Goodman (1981a). Kumar and Hsu (1998) compiles some articles related to recovery in distributed databases. Elections in distributed systems are discussed in Garcia-Molina (1982). Lamport (1978) discusses problems with generating unique timestamps in a distributed system. Rahimi and Haug (2007) discuss a more flexible way to construct query critical metadata for P2P databases. Ouzzani and Bouguettaya (2004) outline fundamental problems in distributed query processing over Web-based data sources.

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) discuss 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 work that discusses replicated data includes Gladney (1989), Agrawal and ElAbbadi (1990), ElAbbadi and Toueg (1989), Kumar and Segev (1993), Mukkamala (1989), and Wolfson and Milo (1991). Bassiouni (1988) discusses optimistic protocols for DDB concurrency control. Garcia-Molina (1983) and Kumar and Stonebraker (1987) discuss techniques that use 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. In more recent work, Vadivelu et al. (2008) propose using backup mechanism and multilevel security to develop algorithms for improving concurrency. Madria et al. (2007) propose a mechanism based on a multiversion two-phase locking scheme and timestamping to address concurrency issues specific to mobile database systems. Boukerche and Tuck (2001) propose a technique that allows transactions to be out of order to a limited extent. They attempt to ease the load on the application developer by exploiting the network environment and producing a schedule equivalent to a temporally ordered serial schedule. Han et al. (2004) propose a deadlock-free and serializable extended Petri net model for Web-based distributed real-time databases.

A survey of recovery techniques in distributed systems is given by Kohler (1981). Reed (1983) discusses atomic actions on distributed data. Bhargava (1987) presents an edited compilation of various approaches and techniques for concurrency and reliability in distributed systems.

Federated database systems were first defined in McLeod and Heimbigner (1985). Techniques for schema integration in federated databases are presented by Elmasri et al. (1986), Batini et al. (1987), Hayne and Ram (1990), and Motro (1987).

Elmagarmid and Helal (1988) and Gamal-Eldin et al. (1988) discuss the update problem in heterogeneous DDBSs. Heterogeneous distributed database issues are discussed in Hsiao and Kamel (1989). Sheth and Larson (1990) present an exhaustive survey of federated database management.

Since late 1980s 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). Castano et al. (1998) present an excellent survey of techniques for analysis of schemas. Pitoura et al. (1995) discuss object orientation in multidatabase systems. Xiao et al. (2003) propose an XML-based model for a common data model for multidatabase systems and present a new approach for schema mapping based on this model. Lakshmanan et al. (2001) propose extending SQL for interoperability and describe the architecture and algorithms for achieving the same.

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. Elmagarmid (1992) discuss transaction processing for advanced applications, including engineering applications discussed in Heiler et al. (1992).

The workflow systems, which are becoming popular to manage information in complex organizations, use multilevel and nested transactions in conjunction with distributed databases. Weikum (1991) discusses multilevel transaction management. Alonso et al. (1997) discuss limitations of current workflow systems. Lopes et al. (2009) propose that users define and execute their own workflows using a client-side Web browser. They attempt to leverage Web 2.0 trends to simplify the user's work for workflow management. Jung and Yeom (2008) exploit data workflow to develop an improved transaction management system that provides simultaneous, transparent access to the heterogeneous storages that constitute the HVEM DataGrid. Deelman and Chervanak (2008) list the challenges in data-intensive scientific workflows. Specifically, they look at automated management of data, efficient mapping techniques, and user feedback issues in workflow mapping. They also argue for data reuse as an efficient means to manage data and present the challenges therein.

A number of experimental distributed DBMSs have been implemented. These include distributed INGRES by Epstein et al., (1978), DDTS by Devor and Weeldreyer, (1980), SDD-1 by Rothnie et al., (1980), System R* by Lindsay et al., (1984), SIRIUS-DELTA by Ferrier and Stangret, (1982), and MULTIBASE by Smith et al., (1981). The OMNIBASE system by Rusinkiewicz et al. (1988) and the Federated Information Base developed using the Candide data model by Navathe et al. (1994) are examples of federated DDBMSs. Pitoura et al. (1995) present a comparative survey of the federated database system prototypes. Most commercial DBMS vendors have products using the client-server approach and offer distributed versions of their systems. Some system issues concerning client-server DBMS architectures are discussed in Carey et al. (1991), DeWitt et al. (1990), and Wang and Rowe (1991). Khoshafian et al. (1992) discuss design issues for relational DBMSs in

the client-server environment. Client-server management issues are discussed in many books, such as Zantinge and Adriaans (1996). Di Stefano (2005) discusses data distribution issues specific to grid computing. A major part of this discussion may also apply to cloud computing.

part 11

Advanced Database Models, Systems, and Applications

chapter 26

Enhanced Data Models for Advanced Applications

As the use of database systems has grown, users have demanded additional functionality from these software packages, with the purpose of making it easier to implement more advanced and complex user applications. Object-oriented databases and object-relational systems do provide features that allow users to extend their systems by specifying additional abstract data types for each application. However, it is quite useful to identify certain common features for some of these advanced applications and to create models that can represent them. Additionally, specialized storage structures and indexing methods can be implemented to improve the performance of these common features. Then the features can be implemented as abstract data types or class libraries and purchased separately from the basic DBMS software package. The term **data blade** has been used in Informix and **cartridge** in Oracle to refer to such optional submodules that can be included in a DBMS package. Users can utilize these features directly if they are suitable for their applications, without having to reinvent, reimplement, and reprogram such common features.

This chapter introduces database concepts for some of the common features that are needed by advanced applications and are being used widely. We will cover *active rules* that are used in active database applications, *temporal concepts* that are used in temporal database applications, and, briefly, some of the issues involving *spatial databases* and *multimedia databases*. We will also discuss *deductive databases*. It is important to note that each of these topics is very broad, and we give only a brief introduction to each. In fact, each of these areas can serve as the sole topic of a complete book.

In Section 26.1 we introduce the topic of active databases, which provide additional functionality for specifying **active rules**. These rules can be automatically triggered

by events that occur, such as database updates or certain times being reached, and can initiate certain actions that have been specified in the rule declaration to occur if certain conditions are met. Many commercial packages include some of the functionality provided by active databases in the form of **triggers**. Triggers are now part of the SQL-99 and later standards.

In Section 26.2 we introduce the concepts of **temporal databases**, which permit the database system to store a history of changes, and allow users to query both current and past states of the database. Some temporal database models also allow users to store future expected information, such as planned schedules. It is important to note that many database applications are temporal, but they are often implemented without having much temporal support from the DBMS package—that is, the temporal concepts are implemented in the application programs that access the database.

Section 26.3 gives a brief overview of **spatial database** concepts. We discuss types of spatial data, different kinds of spatial analyses, operations on spatial data, types of spatial queries, spatial data indexing, spatial data mining, and applications of spatial databases.

Section 26.4 is devoted to multimedia database concepts. **Multimedia databases** provide features that allow users to store and query different types of multimedia information, which includes **images** (such as pictures and drawings), **video clips** (such as movies, newsreels, and home videos), **audio clips** (such as songs, phone messages, and speeches), and **documents** (such as books and articles). We discuss automatic analysis of images, object recognition in images, and semantic tagging of images,

In Section 26.5 we discuss deductive databases,[1] an area that is at the intersection of databases, logic, and artificial intelligence or knowledge bases. A **deductive database system** includes capabilities to define (**deductive**) **rules**, which can deduce or infer additional information from the facts that are stored in a database. Because part of the theoretical foundation for some deductive database systems is mathematical logic, such rules are often referred to as **logic databases**. Other types of systems, referred to as **expert database systems** or **knowledge-based systems**, also incorporate reasoning and inferencing capabilities; such systems use techniques that were developed in the field of artificial intelligence, including semantic networks, frames, production systems, or rules for capturing domain-specific knowledge. Section 26.6 summarizes the chapter.

Readers may choose to peruse the particular topics they are interested in, as the sections in this chapter are practically independent of one another.

[1]Section 26.5 is a summary of Deductive Databases. The full chapter from the third edition, which provides a more comprehensive introduction, is available on the book's Web site.

26.1 Active Database Concepts and Triggers

Rules that specify actions that are automatically triggered by certain events have been considered important enhancements to database systems for quite some time. In fact, the concept of **triggers**—a technique for specifying certain types of active rules—has existed in early versions of the SQL specification for relational databases and triggers are now part of the SQL-99 and later standards. Commercial relational DBMSs—such as Oracle, DB2, and Microsoft SQLServer—have various versions of triggers available. However, much research into what a general model for active databases should look like has been done since the early models of triggers were proposed. In Section 26.1.1 we will present the general concepts that have been proposed for specifying rules for active databases. We will use the syntax of the Oracle commercial relational DBMS to illustrate these concepts with specific examples, since Oracle triggers are close to the way rules are specified in the SQL standard. Section 26.1.2 will discuss some general design and implementation issues for active databases. We give examples of how active databases are implemented in the STARBURST experimental DBMS in Section 26.1.3, since STARBURST provides for many of the concepts of generalized active databases within its framework. Section 26.1.4 discusses possible applications of active databases. Finally, Section 26.1.5 describes how triggers are declared in the SQL-99 standard.

26.1.1 Generalized Model for Active Databases and Oracle Triggers

The model that has been used to specify active database rules is referred to as the **Event-Condition-Action** (**ECA**) model. A rule in the ECA model has three components:

1. The **event(s)** that triggers the rule: These events are usually database update operations that are explicitly applied to the database. However, in the general model, they could also be temporal events[2] or other kinds of external events.
2. The **condition** that determines whether the rule action should be executed: Once the triggering event has occurred, an *optional* condition may be evaluated. If *no condition* is specified, the action will be executed once the event occurs. If a condition is specified, it is first evaluated, and only *if it evaluates to true* will the rule action be executed.
3. The **action** to be taken: The action is usually a sequence of SQL statements, but it could also be a database transaction or an external program that will be automatically executed.

Let us consider some examples to illustrate these concepts. The examples are based on a much simplified variation of the COMPANY database application from Figure 3.5 and is shown in Figure 26.1, with each employee having a name (Name), Social

[2]An example would be a temporal event specified as a periodic time, such as: Trigger this rule every day at 5:30 A.M.

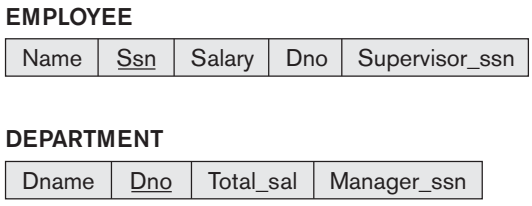


Figure 26.1
A simplified COMPANY database used for active rule examples.

Security number (Ssn), salary (Salary), department to which they are currently assigned (Dno, a foreign key to DEPARTMENT), and a direct supervisor (Supervisor_ssn, a (recursive) foreign key to EMPLOYEE). For this example, we assume that NULL is allowed for Dno, indicating that an employee may be temporarily unassigned to any department. Each department has a name (Dname), number (Dno), the total salary of all employees assigned to the department (Total_sal), and a manager (Manager_ssn, which is a foreign key to EMPLOYEE).

Notice that the Total_sal attribute is really a derived attribute, whose value should be the sum of the salaries of all employees who are assigned to the particular department. Maintaining the correct value of such a derived attribute can be done via an active rule. First we have to determine the **events** that *may cause* a change in the value of Total_sal, which are as follows:

1. Inserting (one or more) new employee tuples
2. Changing the salary of (one or more) existing employees
3. Changing the assignment of existing employees from one department to another
4. Deleting (one or more) employee tuples

In the case of event 1, we only need to recompute Total_sal if the new employee is immediately assigned to a department—that is, if the value of the Dno attribute for the new employee tuple is not NULL (assuming NULL is allowed for Dno). Hence, this would be the **condition** to be checked. A similar condition could be checked for event 2 (and 4) to determine whether the employee whose salary is changed (or who is being deleted) is currently assigned to a department. For event 3, we will always execute an action to maintain the value of Total_sal correctly, so no condition is needed (the action is always executed).

The **action** for events 1, 2, and 4 is to automatically update the value of Total_sal for the employee's department to reflect the newly inserted, updated, or deleted employee's salary. In the case of event 3, a twofold action is needed: one to update the Total_sal of the employee's old department and the other to update the Total_sal of the employee's new department.

The four active rules (or triggers) R1, R2, R3, and R4—corresponding to the above situation—can be specified in the notation of the Oracle DBMS as shown in Figure 26.2(a). Let us consider rule R1 to illustrate the syntax of creating triggers in Oracle.

(a)
```
R1: CREATE TRIGGER Total_sal1
    AFTER INSERT ON EMPLOYEE
    FOR EACH ROW
    WHEN ( NEW.Dno IS NOT NULL )
        UPDATE DEPARTMENT
        SET Total_sal = Total_sal + NEW.Salary
        WHERE Dno = NEW.Dno;

R2: CREATE TRIGGER Total_sal2
    AFTER UPDATE OF Salary ON EMPLOYEE
    FOR EACH ROW
    WHEN ( NEW.Dno IS NOT NULL )
        UPDATE DEPARTMENT
        SET Total_sal = Total_sal + NEW.Salary – OLD.Salary
        WHERE Dno = NEW.Dno;

R3: CREATE TRIGGER Total_sal3
    AFTER UPDATE OF Dno ON EMPLOYEE
    FOR EACH ROW
        BEGIN
        UPDATE DEPARTMENT
        SET Total_sal = Total_sal + NEW.Salary
        WHERE Dno = NEW.Dno;
        UPDATE DEPARTMENT
        SET Total_sal = Total_sal – OLD.Salary
        WHERE Dno = OLD.Dno;
        END;

R4: CREATE TRIGGER Total_sal4
    AFTER DELETE ON EMPLOYEE
    FOR EACH ROW
    WHEN ( OLD.Dno IS NOT NULL )
        UPDATE DEPARTMENT
        SET Total_sal = Total_sal – OLD.Salary
        WHERE Dno = OLD.Dno;
```

(b)
```
R5: CREATE TRIGGER Inform_supervisor1
    BEFORE INSERT OR UPDATE OF Salary, Supervisor_ssn
        ON EMPLOYEE
    FOR EACH ROW
    WHEN ( NEW.Salary > ( SELECT Salary FROM EMPLOYEE
                          WHERE Ssn = NEW.Supervisor_ssn ) )
            inform_supervisor(NEW.Supervisor_ssn, NEW.Ssn );
```

Figure 26.2
Specifying active rules as triggers in Oracle notation. (a) Triggers for automatically maintaining the consistency of Total_sal of DEPARTMENT. (b) Trigger for comparing an employee's salary with that of his or her supervisor.

The CREATE TRIGGER statement specifies a trigger (or active rule) name—Total_sal1 for R1. The AFTER clause specifies that the rule will be triggered *after* the events that trigger the rule occur. The triggering events—an insert of a new employee in this example—are specified following the AFTER keyword.[3]

The ON clause specifies the relation on which the rule is specified—EMPLOYEE for R1. The *optional* keywords FOR EACH ROW specify that the rule will be triggered *once for each row* that is affected by the triggering event.[4]

The *optional* WHEN clause is used to specify any conditions that need to be checked after the rule is triggered, but before the action is executed. Finally, the action(s) to be taken is (are) specified as a PL/SQL block, which typically contains one or more SQL statements or calls to execute external procedures.

The four triggers (active rules) R1, R2, R3, and R4 illustrate a number of features of active rules. First, the basic **events** that can be specified for triggering the rules are the standard SQL update commands: INSERT, DELETE, and UPDATE. They are specified by the keywords **INSERT**, **DELETE**, and **UPDATE** in Oracle notation. In the case of UPDATE, one may specify the attributes to be updated—for example, by writing **UPDATE OF** Salary, Dno. Second, the rule designer needs to have a way to refer to the tuples that have been inserted, deleted, or modified by the triggering event. The keywords **NEW** and **OLD** are used in Oracle notation; NEW is used to refer to a newly inserted or newly updated tuple, whereas OLD is used to refer to a deleted tuple or to a tuple before it was updated.

Thus, rule R1 is triggered after an INSERT operation is applied to the EMPLOYEE relation. In R1, the condition (**NEW**.Dno **IS NOT NULL**) is checked, and if it evaluates to true, meaning that the newly inserted employee tuple is related to a department, then the action is executed. The action updates the DEPARTMENT tuple(s) related to the newly inserted employee by adding their salary (**NEW**.Salary) to the Total_sal attribute of their related department.

Rule R2 is similar to R1, but it is triggered by an UPDATE operation that updates the SALARY of an employee rather than by an INSERT. Rule R3 is triggered by an update to the Dno attribute of EMPLOYEE, which signifies changing an employee's assignment from one department to another. There is no condition to check in R3, so the action is executed whenever the triggering event occurs. The action updates both the old department and new department of the reassigned employees by adding their salary to Total_sal of their *new* department and subtracting their salary from Total_sal of their *old* department. Note that this should work even if the value of Dno is NULL, because in this case no department will be selected for the rule action.[5]

---

[3]As we will see, it is also possible to specify BEFORE instead of AFTER, which indicates that the rule is triggered *before the triggering event is executed.*

[4]Again, we will see that an alternative is to trigger the rule *only once* even if multiple rows (tuples) are affected by the triggering event.

[5]R1, R2, and R4 can also be written without a condition. However, it may be more efficient to execute them with the condition since the action is not invoked unless it is required.

It is important to note the effect of the optional FOR EACH ROW clause, which signifies that the rule is triggered separately *for each tuple.* This is known as a **row-level trigger**. If this clause was left out, the trigger would be known as a **statement-level trigger** and would be triggered once for each triggering statement. To see the difference, consider the following update operation, which gives a 10 percent raise to all employees assigned to department 5. This operation would be an event that triggers rule R2:

```
UPDATE  EMPLOYEE
SET     Salary = 1.1 * Salary
WHERE   Dno = 5;
```

Because the above statement could update multiple records, a rule using row-level semantics, such as R2 in Figure 26.2, would be triggered *once for each row,* whereas a rule using statement-level semantics is triggered *only once.* The Oracle system allows the user to choose which of the above options is to be used for each rule. Including the optional FOR EACH ROW clause creates a row-level trigger, and leaving it out creates a statement-level trigger. Note that the keywords NEW and OLD can only be used with row-level triggers.

As a second example, suppose we want to check whenever an employee's salary is greater than the salary of his or her direct supervisor. Several events can trigger this rule: inserting a new employee, changing an employee's salary, or changing an employee's supervisor. Suppose that the action to take would be to call an external procedure inform_supervisor,[6] which will notify the supervisor. The rule could then be written as in R5 (see Figure 26.2(b)).

Figure 26.3 shows the syntax for specifying some of the main options available in Oracle triggers. We will describe the syntax for triggers in the SQL-99 standard in Section 26.1.5.

Figure 26.3
A syntax summary for specifying triggers in the Oracle system (main options only).

```
<trigger>            ::= CREATE TRIGGER <trigger name>
                         ( AFTER | BEFORE ) <triggering events> ON <table name>
                         [ FOR EACH ROW ]
                         [ WHEN <condition> ]
                         <trigger actions> ;
<triggering events>  ::= <trigger event> {OR <trigger event> }
<trigger event>      ::= INSERT | DELETE | UPDATE [ OF <column name> { , <column name> } ]
<trigger action>     ::= <PL/SQL block>
```

[6]Assuming that an appropriate external procedure has been declared. This is a feature that is available in SQL-99 and later standards.

26.1.2 Design and Implementation Issues for Active Databases

The previous section gave an overview of some of the main concepts for specifying active rules. In this section, we discuss some additional issues concerning how rules are designed and implemented. The first issue concerns activation, deactivation, and grouping of rules. In addition to creating rules, an active database system should allow users to *activate, deactivate,* and *drop* rules by referring to their rule names. A **deactivated rule** will not be triggered by the triggering event. This feature allows users to selectively deactivate rules for certain periods of time when they are not needed. The **activate command** will make the rule active again. The **drop command** deletes the rule from the system. Another option is to group rules into named **rule sets**, so the whole set of rules can be activated, deactivated, or dropped. It is also useful to have a command that can trigger a rule or rule set via an explicit **PROCESS RULES** command issued by the user.

The second issue concerns whether the triggered action should be executed *before, after, instead of*, or *concurrently with* the triggering event. A **before trigger** executes the trigger before executing the event that caused the trigger. It can be used in applications such as checking for constraint violations. An **after trigger** executes the trigger after executing the event, and it can be used in applications such as maintaining derived data and monitoring for specific events and conditions. An **instead of trigger** executes the trigger instead of executing the event, and it can be used in applications such as executing corresponding updates on base relations in response to an event that is an update of a view.

A related issue is whether the action being executed should be considered as a *separate transaction* or whether it should be part of the same transaction that triggered the rule. We will try to categorize the various options. It is important to note that not all options may be available for a particular active database system. In fact, most commercial systems are *limited to one or two of the options* that we will now discuss.

Let us assume that the triggering event occurs as part of a transaction execution. We should first consider the various options for how the triggering event is related to the evaluation of the rule's condition. The rule *condition evaluation* is also known as **rule consideration**, since the action is to be executed only after considering whether the condition evaluates to true or false. There are three main possibilities for rule consideration:

1. **Immediate consideration.** The condition is evaluated as part of the same transaction as the triggering event, and is evaluated *immediately.* This case can be further categorized into three options:
   - Evaluate the condition *before* executing the triggering event.
   - Evaluate the condition *after* executing the triggering event.
   - Evaluate the condition *instead of* executing the triggering event.
2. **Deferred consideration.** The condition is evaluated at the end of the transaction that included the triggering event. In this case, there could be many triggered rules waiting to have their conditions evaluated.

3. **Detached consideration.** The condition is evaluated as a separate transaction, spawned from the triggering transaction.

The next set of options concerns the relationship between evaluating the rule condition and *executing* the rule action. Here, again, three options are possible: **immediate**, **deferred**, or **detached** execution. Most active systems use the first option. That is, as soon as the condition is evaluated, if it returns true, the action is *immediately* executed.

The Oracle system (see Section 26.1.1) uses the *immediate consideration* model, but it allows the user to specify for each rule whether the *before* or *after* option is to be used with immediate condition evaluation. It also uses the *immediate execution* model. The STARBURST system (see Section 26.1.3) uses the *deferred consideration* option, meaning that all rules triggered by a transaction wait until the triggering transaction reaches its end and issues its COMMIT WORK command before the rule conditions are evaluated.[7]

Another issue concerning active database rules is the distinction between *row-level rules* and *statement-level rules.* Because SQL update statements (which act as triggering events) can specify a set of tuples, one has to distinguish between whether the rule should be considered once for the *whole statement* or whether it should be considered separately *for each row* (that is, tuple) affected by the statement. The SQL-99 standard (see Section 26.1.5) and the Oracle system (see Section 26.1.1) allow the user to choose which of the options is to be used for each rule, whereas STARBURST uses statement-level semantics only. We will give examples of how statement-level triggers can be specified in Section 26.1.3.

One of the difficulties that may have limited the widespread use of active rules, in spite of their potential to simplify database and software development, is that there are no easy-to-use techniques for designing, writing, and verifying rules. For example, it is quite difficult to verify that a set of rules is **consistent**, meaning that two or more rules in the set do not contradict one another. It is also difficult to guarantee **termination** of a set of rules under all circumstances. To illustrate the termination

```
R1: CREATE TRIGGER T1
    AFTER INSERT ON TABLE1
    FOR EACH ROW
        UPDATE TABLE2
        SET Attribute1 = ... ;

R2: CREATE TRIGGER T2
    AFTER UPDATE OF Attribute1 ON TABLE2
    FOR EACH ROW
        INSERT INTO TABLE1 VALUES ( ... );
```

Figure 26.4
An example to illustrate the termination problem for active rules.

[7]STARBURST also allows the user to start rule consideration explicitly via a PROCESS RULES command.

problem briefly, consider the rules in Figure 26.4. Here, rule R1 is triggered by an INSERT event on TABLE1 and its action includes an update event on Attribute1 of TABLE2. However, rule R2's triggering event is an UPDATE event on Attribute1 of TABLE2, and its action includes an INSERT event on TABLE1. In this example, it is easy to see that these two rules can trigger one another indefinitely, leading to nontermination. However, if dozens of rules are written, it is very difficult to determine whether termination is guaranteed or not.

If active rules are to reach their potential, it is necessary to develop tools for the design, debugging, and monitoring of active rules that can help users design and debug their rules.

26.1.3 Examples of Statement-Level Active Rules in STARBURST

We now give some examples to illustrate how rules can be specified in the STARBURST experimental DBMS. This will allow us to demonstrate how statement-level rules can be written, since these are the only types of rules allowed in STARBURST.

The three active rules R1S, R2S, and R3S in Figure 26.5 correspond to the first three rules in Figure 26.2, but they use STARBURST notation and statement-level semantics. We can explain the rule structure using rule R1S. The CREATE RULE statement specifies a rule name—Total_sal1 for R1S. The ON clause specifies the relation on which the rule is specified—EMPLOYEE for R1S. The WHEN clause is used to specify the **events** that trigger the rule.[8] The *optional* IF clause is used to specify any **conditions** that need to be checked. Finally, the THEN clause is used to specify the **actions** to be taken, which are typically one or more SQL statements.

In STARBURST, the basic events that can be specified for triggering the rules are the standard SQL update commands: INSERT, DELETE, and UPDATE. These are specified by the keywords **INSERTED**, **DELETED**, and **UPDATED** in STARBURST notation. Second, the rule designer needs to have a way to refer to the tuples that have been modified. The keywords **INSERTED**, **DELETED**, **NEW-UPDATED**, and **OLD-UPDATED** are used in STARBURST notation to refer to four **transition tables** (relations) that include the newly inserted tuples, the deleted tuples, the updated tuples *before* they were updated, and the updated tuples *after* they were updated, respectively. Obviously, depending on the triggering events, only some of these transition tables may be available. The rule writer can refer to these tables when writing the condition and action parts of the rule. Transition tables contain tuples of the same type as those in the relation specified in the ON clause of the rule—for R1S, R2S, and R3S, this is the EMPLOYEE relation.

In statement-level semantics, the rule designer can only refer to the transition tables as a whole and the rule is triggered only once, so the rules must be written differently than for row-level semantics. Because multiple employee tuples may be

[8]Note that the WHEN keyword specifies *events* in STARBURST but is used to specify the rule *condition* in SQL and Oracle triggers.

```
R1S:  CREATE RULE Total_sal1 ON EMPLOYEE
      WHEN  INSERTED
      IF    EXISTS   ( SELECT * FROM INSERTED WHERE Dno IS NOT NULL )
      THEN  UPDATE     DEPARTMENT AS D
            SET        D.Total_sal = D.Total_sal +
                       ( SELECT SUM (I.Salary) FROM INSERTED AS I WHERE D.Dno = I.Dno )
            WHERE      D.Dno IN ( SELECT Dno FROM INSERTED );

R2S:  CREATE RULE Total_sal2 ON EMPLOYEE
      WHEN  UPDATED    ( Salary )
      IF    EXISTS     ( SELECT * FROM NEW-UPDATED WHERE Dno IS NOT NULL )
            OR EXISTS  ( SELECT * FROM OLD-UPDATED WHERE Dno IS NOT NULL )
      THEN  UPDATE     DEPARTMENT AS D
            SET        D.Total_sal = D.Total_sal +
                       ( SELECT SUM (N.Salary) FROM NEW-UPDATED AS N
                         WHERE D.Dno = N.Dno ) –
                       ( SELECT SUM (O.Salary) FROM OLD-UPDATED AS O
                         WHERE D.Dno = O.Dno )
            WHERE      D.Dno IN ( SELECT Dno FROM NEW-UPDATED ) OR
                       D.Dno IN ( SELECT Dno FROM OLD-UPDATED );

R3S:  CREATE RULE Total_sal3 ON EMPLOYEE
      WHEN  UPDATED    ( Dno )
      THEN  UPDATE     DEPARTMENT AS D
            SET        D.Total_sal = D.Total_sal +
                       ( SELECT SUM (N.Salary) FROM NEW-UPDATED AS N
                         WHERE D.Dno = N.Dno )
            WHERE      D.Dno IN ( SELECT Dno FROM NEW-UPDATED );
            UPDATE     DEPARTMENT AS D
            SET        D.Total_sal = Total_sal –
                       ( SELECT SUM (O.Salary) FROM OLD-UPDATED AS O
                         WHERE D.Dno = O.Dno )
            WHERE      D.Dno IN ( SELECT Dno FROM OLD-UPDATED );
```

Figure 26.5
Active rules using statement-level semantics in STARBURST notation.

inserted in a single insert statement, we have to check if *at least one* of the newly inserted employee tuples is related to a department. In R1S, the condition

EXISTS (**SELECT** * **FROM INSERTED WHERE** Dno **IS NOT NULL**)

is checked, and if it evaluates to true, then the action is executed. The action updates in a single statement the DEPARTMENT tuple(s) related to the newly inserted employee(s) by adding their salaries to the Total_sal attribute of each related department. Because more than one newly inserted employee may belong to the same

department, we use the SUM aggregate function to ensure that all their salaries are added.

Rule R2S is similar to R1S, but is triggered by an UPDATE operation that updates the salary of one or more employees rather than by an INSERT. Rule R3S is triggered by an update to the Dno attribute of EMPLOYEE, which signifies changing one or more employees' assignment from one department to another. There is no condition in R3S, so the action is executed whenever the triggering event occurs.[9] The action updates both the old department(s) and new department(s) of the reassigned employees by adding their salary to Total_sal of each *new* department and subtracting their salary from Total_sal of each *old* department.

In our example, it is more complex to write the statement-level rules than the row-level rules, as can be illustrated by comparing Figures 26.2 and 26.5. However, this is not a general rule, and other types of active rules may be easier to specify when using statement-level notation than when using row-level notation.

The execution model for active rules in STARBURST uses **deferred consideration**. That is, all the rules that are triggered within a transaction are placed in a set—called the **conflict set**—which is not considered for evaluation of conditions and execution until the transaction ends (by issuing its COMMIT WORK command). STARBURST also allows the user to explicitly start rule consideration in the middle of a transaction via an explicit PROCESS RULES command. Because multiple rules must be evaluated, it is necessary to specify an order among the rules. The syntax for rule declaration in STARBURST allows the specification of *ordering* among the rules to instruct the system about the order in which a set of rules should be considered.[10] Additionally, the transition tables—INSERTED, DELETED, NEW-UPDATED, and OLD-UPDATED—contain the *net effect* of all the operations within the transaction that affected each table, since multiple operations may have been applied to each table during the transaction.

26.1.4 Potential Applications for Active Databases

We now briefly discuss some of the potential applications of active rules. Obviously, one important application is to allow **notification** of certain conditions that occur. For example, an active database may be used to monitor, say, the temperature of an industrial furnace. The application can periodically insert in the database the temperature reading records directly from temperature sensors, and active rules can be written that are triggered whenever a temperature record is inserted, with a condition that checks if the temperature exceeds the danger level, and results in the action to raise an alarm.

---

[9] As in the Oracle examples, rules R1S and R2S can be written without a condition. However, it may be more efficient to execute them with the condition since the action is not invoked unless it is required.

[10] If no order is specified between a pair of rules, the system default order is based on placing the rule declared first ahead of the other rule.

Active rules can also be used to **enforce integrity constraints** by specifying the types of events that may cause the constraints to be violated and then evaluating appropriate conditions that check whether the constraints are actually violated by the event or not. Hence, complex application constraints, often known as **business rules**, may be enforced that way. For example, in the UNIVERSITY database application, one rule may monitor the GPA of students whenever a new grade is entered, and it may alert the advisor if the GPA of a student falls below a certain threshold; another rule may check that course prerequisites are satisfied before allowing a student to enroll in a course; and so on.

Other applications include the automatic **maintenance of derived data**, such as the examples of rules R1 through R4 that maintain the derived attribute Total_sal whenever individual employee tuples are changed. A similar application is to use active rules to maintain the consistency of **materialized views** (see Section 5.3) whenever the base relations are modified. Alternately, an update operation specified on a view can be a triggering event, which can be converted to updates on the base relations by using an *instead of* trigger. These applications are also relevant to the new data warehousing technologies (see Chapter 29). A related application maintains that **replicated tables** are consistent by specifying rules that modify the replicas whenever the master table is modified.

26.1.5 Triggers in SQL-99

Triggers in the SQL-99 and later standards are quite similar to the examples we discussed in Section 26.1.1, with some minor syntactic differences. The basic **events** that can be specified for triggering the rules are the standard SQL update commands: INSERT, DELETE, and UPDATE. In the case of UPDATE, one may specify the attributes to be updated. Both row-level and statement-level triggers are allowed, indicated in the trigger by the clauses FOR EACH ROW and FOR EACH STATEMENT, respectively. One syntactic difference is that the trigger may specify particular tuple variable names for the old and new tuples instead of using the keywords NEW and OLD, as shown in Figure 26.1. Trigger T1 in Figure 26.6 shows how the row-level trigger R2 from Figure 26.1(a) may be specified in SQL-99. Inside the REFERENCING clause, we named tuple variables (aliases) O and N to refer to the OLD tuple (before modification) and NEW tuple (after modification), respectively. Trigger T2 in Figure 26.6 shows how the statement-level trigger R2S from Figure 26.5 may be specified in SQL-99. For a statement-level trigger, the REFERENCING clause is used to refer to the table of all new tuples (newly inserted or newly updated) as N, whereas the table of all old tuples (deleted tuples or tuples before they were updated) is referred to as O.

26.2 Temporal Database Concepts

Temporal databases, in the broadest sense, encompass all database applications that require some aspect of time when organizing their information. Hence, they provide a good example to illustrate the need for developing a set of unifying concepts for application developers to use. Temporal database applications have been

```
T1: CREATE TRIGGER Total_sal1
    AFTER UPDATE OF Salary ON EMPLOYEE
    REFERENCING OLD ROW AS O, NEW ROW AS N
    FOR EACH ROW
    WHEN  ( N.Dno IS NOT NULL )
    UPDATE DEPARTMENT
    SET Total_sal = Total_sal + N.salary – O.salary
    WHERE Dno = N.Dno;

T2: CREATE TRIGGER Total_sal2
    AFTER UPDATE OF Salary ON EMPLOYEE
    REFERENCING OLD TABLE AS O, NEW TABLE AS N
    FOR EACH STATEMENT
    WHEN  EXISTS ( SELECT *FROM N WHERE N.Dno IS NOT NULL ) OR
          EXISTS ( SELECT * FROM O WHERE O.Dno IS NOT NULL )
    UPDATE DEPARTMENT AS D
    SET D.Total_sal = D.Total_sal
    + ( SELECT SUM (N.Salary) FROM N WHERE D.Dno=N.Dno )
    – ( SELECT SUM (O.Salary) FROM O WHERE D.Dno=O.Dno )
    WHERE Dno IN ( ( SELECT Dno FROM N ) UNION ( SELECT Dno FROM O ) );
```

Figure 26.6
Trigger T1 illustrating the syntax for defining triggers in SQL-99.

developed since the early days of database usage. However, in creating these applications, it is mainly left to the application designers and developers to discover, design, program, and implement the temporal concepts they need. There are many examples of applications where some aspect of time is needed to maintain the information in a database. These include *healthcare,* where patient histories need to be maintained; *insurance,* where claims and accident histories are required as well as information about the times when insurance policies are in effect; *reservation systems* in general (hotel, airline, car rental, train, and so on), where information on the dates and times when reservations are in effect are required; *scientific databases,* where data collected from experiments includes the time when each data is measured; and so on. Even the two examples used in this book may be easily expanded into temporal applications. In the COMPANY database, we may wish to keep SALARY, JOB, and PROJECT histories on each employee. In the UNIVERSITY database, time is already included in the SEMESTER and YEAR of each SECTION of a COURSE, the grade history of a STUDENT, and the information on research grants. In fact, it is realistic to conclude that the majority of database applications have some temporal information. However, users often attempt to simplify or ignore temporal aspects because of the complexity that they add to their applications.

In this section, we will introduce some of the concepts that have been developed to deal with the complexity of temporal database applications. Section 26.2.1 gives an overview of how time is represented in databases, the different types of temporal

information, and some of the different dimensions of time that may be needed. Section 26.2.2 discusses how time can be incorporated into relational databases. Section 26.2.3 gives some additional options for representing time that are possible in database models that allow complex-structured objects, such as object databases. Section 26.2.4 introduces operations for querying temporal databases, and gives a brief overview of the TSQL2 language, which extends SQL with temporal concepts. Section 26.2.5 focuses on time series data, which is a type of temporal data that is very important in practice.

26.2.1 Time Representation, Calendars, and Time Dimensions

For temporal databases, time is considered to be an *ordered sequence* of **points** in some **granularity** that is determined by the application. For example, suppose that some temporal application never requires time units that are less than one second. Then, each time point represents one second using this granularity. In reality, each second is a (short) *time duration,* not a point, since it may be further divided into milliseconds, microseconds, and so on. Temporal database researchers have used the term **chronon** instead of point to describe this minimal granularity for a particular application. The main consequence of choosing a minimum granularity—say, one second—is that events occurring within the same second will be considered to be *simultaneous events,* even though in reality they may not be.

Because there is no known beginning or ending of time, one needs a reference point from which to measure specific time points. Various calendars are used by various cultures (such as Gregorian (western), Chinese, Islamic, Hindu, Jewish, Coptic, and so on) with different reference points. A **calendar** organizes time into different time units for convenience. Most calendars group 60 seconds into a minute, 60 minutes into an hour, 24 hours into a day (based on the physical time of earth's rotation around its axis), and 7 days into a week. Further grouping of days into months and months into years either follow solar or lunar natural phenomena, and are generally irregular. In the Gregorian calendar, which is used in most western countries, days are grouped into months that are 28, 29, 30, or 31 days, and 12 months are grouped into a year. Complex formulas are used to map the different time units to one another.

In SQL2, the temporal data types (see Chapter 4) include DATE (specifying Year, Month, and Day as YYYY-MM-DD), TIME (specifying Hour, Minute, and Second as HH:MM:SS), TIMESTAMP (specifying a Date/Time combination, with options for including subsecond divisions if they are needed), INTERVAL (a relative time duration, such as 10 days or 250 minutes), and PERIOD (an *anchored* time duration with a fixed starting point, such as the 10-day period from January 1, 2009, to January 10, 2009, inclusive).[11]

[11]Unfortunately, the terminology has not been used consistently. For example, the term *interval* is often used to denote an anchored duration. For consistency, we will use the SQL terminology.

Event Information versus Duration (or State) Information. A temporal database will store information concerning when certain events occur, or when certain facts are considered to be true. There are several different types of temporal information. **Point events** or **facts** are typically associated in the database with a **single time point** in some granularity. For example, a bank deposit event may be associated with the timestamp when the deposit was made, or the total monthly sales of a product (fact) may be associated with a particular month (say, February 2010). Note that even though such events or facts may have different granularities, each is still associated with a *single time value* in the database. This type of information is often represented as **time series data** as we will discuss in Section 26.2.5. **Duration events** or **facts**, on the other hand, are associated with a specific **time period** in the database.[12] For example, an employee may have worked in a company from August 15, 2003 until November 20, 2008.

A **time period** is represented by its **start** and **end time points** [START-TIME, END-TIME]. For example, the above period is represented as [2003-08-15, 2008-11-20]. Such a time period is often interpreted to mean the *set of all time points* from start-time to end-time, inclusive, in the specified granularity. Hence, assuming day granularity, the period [2003-08-15, 2008-11-20] represents the set of all days from August 15, 2003, until November 20, 2008, inclusive.[13]

Valid Time and Transaction Time Dimensions. Given a particular event or fact that is associated with a particular time point or time period in the database, the association may be interpreted to mean different things. The most natural interpretation is that the associated time is the time that the event occurred, or the period during which the fact was considered to be true *in the real world.* If this interpretation is used, the associated time is often referred to as the **valid time**. A temporal database using this interpretation is called a **valid time database**.

However, a different interpretation can be used, where the associated time refers to the time when the information was actually stored in the database; that is, it is the value of the system time clock when the information is valid *in the system.*[14] In this case, the associated time is called the **transaction time**. A temporal database using this interpretation is called a **transaction time database**.

Other interpretations can also be intended, but these are considered to be the most common ones, and they are referred to as **time dimensions**. In some applications, only one of the dimensions is needed and in other cases both time dimensions are required, in which case the temporal database is called a **bitemporal database**. If

[12]This is the same as an *anchored duration*. It has also been frequently called a *time interval*, but to avoid confusion we will use *period* to be consistent with SQL terminology.

[13]The representation [2003-08-15, 2008-11-20] is called a *closed interval* representation. One can also use an *open interval*, denoted [2003-08-15, 2008-11-21), where the set of points does not include the end point. Although the latter representation is sometimes more convenient, we shall use closed intervals except where indicated.

[14]The explanation is more involved, as we will see in Section 26.2.3.

other interpretations are intended for time, the user can define the semantics and program the applications appropriately, and it is called a **user-defined time**.

The next section shows how these concepts can be incorporated into relational databases, and Section 26.2.3 shows an approach to incorporate temporal concepts into object databases.

26.2.2 Incorporating Time in Relational Databases Using Tuple Versioning

Valid Time Relations. Let us now see how the different types of temporal databases may be represented in the relational model. First, suppose that we would like to include the history of changes as they occur in the real world. Consider again the database in Figure 26.1, and let us assume that, for this application, the granularity is day. Then, we could convert the two relations EMPLOYEE and DEPARTMENT into **valid time relations** by adding the attributes Vst (Valid Start Time) and Vet (Valid End Time), whose data type is DATE in order to provide day granularity. This is shown in Figure 26.7(a), where the relations have been renamed EMP_VT and DEPT_VT, respectively.

Consider how the EMP_VT relation differs from the nontemporal EMPLOYEE relation (Figure 26.1).[15] In EMP_VT, each tuple V represents a **version** of an employee's

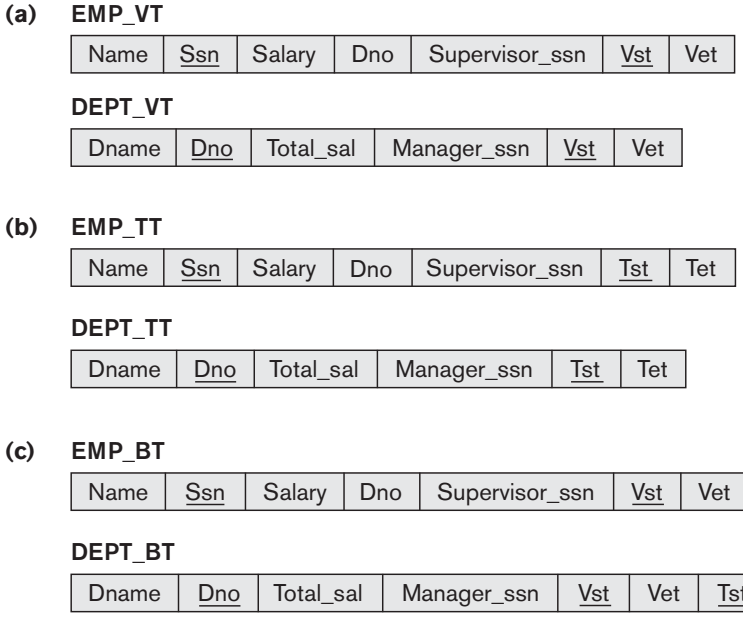


Figure 26.7
Different types of temporal relational databases. (a) Valid time database schema. (b) Transaction time database schema. (c) Bitemporal database schema.

[15]A nontemporal relation is also called a **snapshot relation** because it shows only the *current snapshot* or *current state* of the database.

information that is valid (in the real world) only during the time period [V.Vst, V.Vet], whereas in EMPLOYEE each tuple represents only the current state or current version of each employee. In EMP_VT, the **current version** of each employee typically has a special value, *now,* as its valid end time. This special value, ***now***, is a **temporal variable** that implicitly represents the current time as time progresses. The nontemporal EMPLOYEE relation would only include those tuples from the EMP_VT relation whose Vet is *now.*

Figure 26.8 shows a few tuple versions in the valid-time relations EMP_VT and DEPT_VT. There are two versions of Smith, three versions of Wong, one version of Brown, and one version of Narayan. We can now see how a valid time relation should behave when information is changed. Whenever one or more attributes of an employee are **updated**, rather than actually overwriting the old values, as would happen in a nontemporal relation, the system should create a new version and **close** the current version by changing its Vet to the end time. Hence, when the user issued the command to update the salary of Smith effective on June 1, 2003, to $30000, the second version of Smith was created (see Figure 26.8). At the time of this update, the first version of Smith was the current version, with *now* as its Vet, but after the update *now* was changed to May 31, 2003 (one less than June 1, 2003, in day granularity), to indicate that the version has become a **closed** or **history version** and that the new (second) version of Smith is now the current one.

Figure 26.8
Some tuple versions in the valid time relations EMP_VT and DEPT_VT.

EMP_VT

Name	Ssn	Salary	Dno	Supervisor_ssn	Vst	Vet
Smith	123456789	25000	5	333445555	2002-06-15	2003-05-31
Smith	123456789	30000	5	333445555	2003-06-01	Now
Wong	333445555	25000	4	999887777	1999-08-20	2001-01-31
Wong	333445555	30000	5	999887777	2001-02-01	2002-03-31
Wong	333445555	40000	5	888665555	2002-04-01	Now
Brown	222447777	28000	4	999887777	2001-05-01	2002-08-10
Narayan	666884444	38000	5	333445555	2003-08-01	Now

. . .

DEPT_VT

Dname	Dno	Manager_ssn	Vst	Vet
Research	5	888665555	2001-09-20	2002-03-31
Research	5	333445555	2002-04-01	Now

. . .

It is important to note that in a valid time relation, the user must generally provide the valid time of an update. For example, the salary update of Smith may have been entered in the database on May 15, 2003, at 8:52:12 A.M., say, even though the salary change in the real world is effective on June 1, 2003. This is called a **proactive update**, since it is applied to the database *before* it becomes effective in the real world. If the update is applied to the database *after* it becomes effective in the real world, it is called a **retroactive update**. An update that is applied at the same time as it becomes effective is called a **simultaneous update**.

The action that corresponds to **deleting** an employee in a nontemporal database would typically be applied to a valid time database by *closing the current version* of the employee being deleted. For example, if Smith leaves the company effective January 19, 2004, then this would be applied by changing Vet of the current version of Smith from *now* to 2004-01-19. In Figure 26.8, there is no current version for Brown, because he presumably left the company on 2002-08-10 and was *logically deleted.* However, because the database is temporal, the old information on Brown is still there.

The operation to **insert** a new employee would correspond to *creating the first tuple version* for that employee, and making it the current version, with the Vst being the effective (real world) time when the employee starts work. In Figure 26.7, the tuple on Narayan illustrates this, since the first version has not been updated yet.

Notice that in a valid time relation, the *nontemporal key,* such as Ssn in EMPLOYEE, is no longer unique in each tuple (version). The new relation key for EMP_VT is a combination of the nontemporal key and the valid start time attribute Vst,[16] so we use (Ssn, Vst) as primary key. This is because, at any point in time, there should be *at most one valid version* of each entity. Hence, the constraint that any two tuple versions representing the same entity should have *nonintersecting valid time periods* should hold on valid time relations. Notice that if the nontemporal primary key value may change over time, it is important to have a unique **surrogate key attribute**, whose value never changes for each real-world entity, in order to relate all versions of the same real-world entity.

Valid time relations basically keep track of the history of changes as they become effective in the *real world.* Hence, if all real-world changes are applied, the database keeps a history of the *real-world states* that are represented. However, because updates, insertions, and deletions may be applied retroactively or proactively, there is no record of the actual *database state* at any point in time. If the actual database states are important to an application, then one should use *transaction time relations.*

Transaction Time Relations. In a transaction time database, whenever a change is applied to the database, the actual **timestamp** of the transaction that applied the change (insert, delete, or update) is recorded. Such a database is most useful when changes are applied *simultaneously* in the majority of cases—for example, real-time stock trading or banking transactions. If we convert the nontemporal database in

[16]A combination of the nontemporal key and the valid end time attribute **Vet** could also be used.

Figure 26.1 into a transaction time database, then the two relations EMPLOYEE and DEPARTMENT are converted into **transaction time relations** by adding the attributes Tst (Transaction Start Time) and Tet (Transaction End Time), whose data type is typically TIMESTAMP. This is shown in Figure 26.7(b), where the relations have been renamed EMP_TT and DEPT_TT, respectively.

In EMP_TT, each tuple V represents a *version* of an employee's information that was created at actual time V.Tst and was (logically) removed at actual time V.Tet (because the information was no longer correct). In EMP_TT, the *current version* of each employee typically has a special value, ***uc*** (**Until Changed**), as its transaction end time, which indicates that the tuple represents correct information *until it is changed* by some other transaction.[17] A transaction time database has also been called a **rollback database**,[18] because a user can logically roll back to the actual database state at any past point in time T by retrieving all tuple versions V whose transaction time period [V.Tst, V.Tet] includes time point T.

Bitemporal Relations. Some applications require both valid time and transaction time, leading to **bitemporal relations**. In our example, Figure 26.7(c) shows how the EMPLOYEE and DEPARTMENT nontemporal relations in Figure 26.1 would appear as bitemporal relations EMP_BT and DEPT_BT, respectively. Figure 26.9 shows a few tuples in these relations. In these tables, tuples whose transaction end time Tet is *uc* are the ones representing currently valid information, whereas tuples whose Tet is an absolute timestamp are tuples that were valid until (just before) that timestamp. Hence, the tuples with *uc* in Figure 26.9 correspond to the valid time tuples in Figure 26.7. The transaction start time attribute Tst in each tuple is the timestamp of the transaction that created that tuple.

Now consider how an **update operation** would be implemented on a bitemporal relation. In this model of bitemporal databases,[19] *no attributes are physically changed* in any tuple except for the transaction end time attribute Tet with a value of *uc*.[20] To illustrate how tuples are created, consider the EMP_BT relation. The *current version* V of an employee has *uc* in its Tet attribute and *now* in its Vet attribute. If some attribute—say, Salary—is updated, then the transaction T that performs the update should have two parameters: the new value of Salary and the valid time VT when the new salary becomes effective (in the real world). Assume that VT– is the

[17]The *uc* variable in transaction time relations corresponds to the *now* variable in valid time relations. The semantics are slightly different though.

[18]Here, the term *rollback* does not have the same meaning as *transaction rollback* (see Chapter 23) during recovery, where the transaction updates are *physically undone*. Rather, here the updates can be *logically undone*, allowing the user to examine the database as it appeared at a previous time point.

[19]There have been many proposed temporal database models. We describe specific models here as examples to illustrate the concepts.

[20]Some bitemporal models allow the Vet attribute to be changed also, but the interpretations of the tuples are different in those models.

EMP_BT

Name	Ssn	Salary	Dno	Supervisor_ssn	Vst	Vet	Tst	Tet
Smith	123456789	25000	5	333445555	2002-06-15	Now	2002-06-08, 13:05:58	2003-06-04,08:56:12
Smith	123456789	25000	5	333445555	2002-06-15	2003-05-31	2003-06-04, 08:56:12	uc
Smith	123456789	30000	5	333445555	2003-06-01	Now	2003-06-04, 08:56:12	uc
Wong	333445555	25000	4	999887777	1999-08-20	Now	1999-08-20, 11:18:23	2001-01-07,14:33:02
Wong	333445555	25000	4	999887777	1999-08-20	2001-01-31	2001-01-07, 14:33:02	uc
Wong	333445555	30000	5	999887777	2001-02-01	Now	2001-01-07, 14:33:02	2002-03-28,09:23:57
Wong	333445555	30000	5	999887777	2001-02-01	2002-03-31	2002-03-28, 09:23:57	uc
Wong	333445555	40000	5	888667777	2002-04-01	Now	2002-03-28, 09:23:57	uc
Brown	222447777	28000	4	999887777	2001-05-01	Now	2001-04-27, 16:22:05	2002-08-12,10:11:07
Brown	222447777	28000	4	999887777	2001-05-01	2002-08-10	2002-08-12, 10:11:07	uc
Narayan	666884444	38000	5	333445555	2003-08-01	Now	2003-07-28, 09:25:37	uc

. . .

DEPT_VT

Dname	Dno	Manager_ssn	Vst	Vet	Tst	Tet
Research	5	888665555	2001-09-20	Now	2001-09-15,14:52:12	2001-03-28,09:23:57
Research	5	888665555	2001-09-20	1997-03-31	2002-03-28,09:23:57	uc
Research	5	333445555	2002-04-01	Now	2002-03-28,09:23:57	uc

Figure 26.9
Some tuple versions in the bitemporal relations EMP_BT and DEPT_BT.

time point before VT in the given valid time granularity and that transaction T has a timestamp TS(T). Then, the following physical changes would be applied to the EMP_BT table:

1. Make a copy V_2 of the current version V; set V_2.Vet to VT–, V_2.Tst to TS(T), V_2.Tet to *uc,* and insert V_2 in EMP_BT; V_2 is a copy of the previous current version V *after it is closed* at valid time VT–.
2. Make a copy V_3 of the current version V; set V_3.Vst to VT, V_3.Vet to *now,* V_3.Salary to the new salary value, V_3.Tst to TS(T), V_3.Tet to *uc,* and insert V_3 in EMP_BT; V_3 represents the new current version.
3. Set V.Tet to TS(T) since the current version is no longer representing correct information.

As an illustration, consider the first three tuples V_1, V_2, and V_3 in EMP_BT in Figure 26.9. Before the update of Smith's salary from 25000 to 30000, only V_1 was in EMP_BT and it was the current version and its Tet was *uc.* Then, a transaction T whose timestamp TS(T) is '2003-06-04,08:56:12' updates the salary to 30000 with the effective valid time of '2003-06-01'. The tuple V_2 is created, which is a copy of V_1 except that its Vet is set to '2003-05-31', one day less than the new valid time and its Tst is the timestamp of the updating transaction. The tuple V_3 is also created, which has the new salary, its Vst is set to '2003-06-01', and its Tst is also the timestamp of the updating transaction. Finally, the Tet of V_1 is set to the timestamp of

the updating transaction, '2003-06-04,08:56:12'. Note that this is a *retroactive update,* since the updating transaction ran on June 4, 2003, but the salary change is effective on June 1, 2003.

Similarly, when Wong's salary and department are updated (at the same time) to 30000 and 5, the updating transaction's timestamp is '2001-01-07,14:33:02' and the effective valid time for the update is '2001-02-01'. Hence, this is a *proactive update* because the transaction ran on January 7, 2001, but the effective date was February 1, 2001. In this case, tuple V_4 is logically replaced by V_5 and V_6.

Next, let us illustrate how a **delete operation** would be implemented on a bitemporal relation by considering the tuples V_9 and V_{10} in the EMP_BT relation of Figure 26.9. Here, employee Brown left the company effective August 10, 2002, and the logical delete is carried out by a transaction T with TS(T) = 2002-08-12,10:11:07. Before this, V9 was the current version of Brown, and its Tet was *uc.* The logical delete is implemented by setting V_9.Tet to 2002-08-12,10:11:07 to invalidate it, and creating the *final version* V_{10} for Brown, with its Vet = 2002-08-10 (see Figure 26.9). Finally, an **insert operation** is implemented by creating the *first version* as illustrated by V_{11} in the EMP_BT table.

Implementation Considerations. There are various options for storing the tuples in a temporal relation. One is to store all the tuples in the same table, as shown in Figures 26.8 and 26.9. Another option is to create two tables: one for the currently valid information and the other for the rest of the tuples. For example, in the bitemporal EMP_BT relation, tuples with *uc* for their Tet and *now* for their Vet would be in one relation, the *current table,* since they are the ones currently valid (that is, represent the current snapshot), and all other tuples would be in another relation. This allows the database administrator to have different access paths, such as indexes for each relation, and keeps the size of the current table reasonable. Another possibility is to create a third table for corrected tuples whose Tet is not *uc.*

Another option that is available is to *vertically partition* the attributes of the temporal relation into separate relations so that if a relation has many attributes, a whole new tuple version is created whenever any one of the attributes is updated. If the attributes are updated asynchronously, each new version may differ in only one of the attributes, thus needlessly repeating the other attribute values. If a separate relation is created to contain only the attributes that *always change synchronously,* with the primary key replicated in each relation, the database is said to be in **temporal normal form**. However, to combine the information, a variation of join known as **temporal intersection join** would be needed, which is generally expensive to implement.

It is important to note that bitemporal databases allow a complete record of changes. Even a record of corrections is possible. For example, it is possible that two tuple versions of the same employee may have the same valid time but different attribute values as long as their transaction times are disjoint. In this case, the tuple with the later transaction time is a **correction** of the other tuple version. Even incorrectly entered valid times may be corrected this way. The incorrect state of the data-

base will still be available as a previous database state for querying purposes. A database that keeps such a complete record of changes and corrections is sometimes called an **append-only database**.

26.2.3 Incorporating Time in Object-Oriented Databases Using Attribute Versioning

The previous section discussed the **tuple versioning approach** to implementing temporal databases. In this approach, whenever one attribute value is changed, a whole new tuple version is created, even though all the other attribute values will be identical to the previous tuple version. An alternative approach can be used in database systems that support **complex structured objects**, such as object databases (see Chapter 11) or object-relational systems. This approach is called **attribute versioning**.

In attribute versioning, a single complex object is used to store all the temporal changes of the object. Each attribute that changes over time is called a **time-varying attribute**, and it has its values versioned over time by adding temporal periods to the attribute. The temporal periods may represent valid time, transaction time, or bitemporal, depending on the application requirements. Attributes that do not change over time are called **nontime-varying** and are not associated with the temporal periods. To illustrate this, consider the example in Figure 26.10, which is an attribute-versioned valid time representation of EMPLOYEE using the object definition language (ODL) notation for object databases (see Chapter 11). Here, we assumed that name and Social Security number are nontime-varying attributes, whereas salary, department, and supervisor are time-varying attributes (they may change over time). Each time-varying attribute is represented as a list of tuples <Valid_start_time, Valid_end_time, Value>, ordered by valid start time.

Whenever an attribute is changed in this model, the current attribute version is *closed* and a **new attribute version** for this attribute only is appended to the list. This allows attributes to change asynchronously. The current value for each attribute has *now* for its Valid_end_time. When using attribute versioning, it is useful to include a **lifespan temporal attribute** associated with the whole object whose value is one or more valid time periods that indicate the valid time of existence for the whole object. Logical deletion of the object is implemented by closing the lifespan. The constraint that any time period of an attribute within an object should be a subset of the object's lifespan should be enforced.

For bitemporal databases, each attribute version would have a tuple with five components:

<Valid_start_time, Valid_end_time, Trans_start_time, Trans_end_time, Value>

The object lifespan would also include both valid and transaction time dimensions. Therefore, the full capabilities of bitemporal databases can be available with attribute versioning. Mechanisms similar to those discussed earlier for updating tuple versions can be applied to updating attribute versions.

```
class TEMPORAL_SALARY
{   attribute     Date                Valid_start_time;
    attribute     Date                Valid_end_time;
    attribute     float               Salary;
};

class TEMPORAL_DEPT
{   attribute     Date                Valid_start_time;
    attribute     Date                Valid_end_time;
    attribute     DEPARTMENT_VT       Dept;
};

class TEMPORAL_SUPERVISOR
{   attribute     Date                Valid_start_time;
    attribute     Date                Valid_end_time;
    attribute     EMPLOYEE_VT         Supervisor;
};

class TEMPORAL_LIFESPAN
{   attribute     Date                Valid_ start time;
    attribute     Date                Valid end time;
};

class EMPLOYEE_VT
(   extent EMPLOYEES   )
{   attribute     list<TEMPORAL_LIFESPAN>       lifespan;
    attribute     string                        Name;
    attribute     string                        Ssn;
    attribute     list<TEMPORAL_SALARY>         Sal_history;
    attribute     list<TEMPORAL_DEPT>           Dept_history;
    attribute     list <TEMPORAL_SUPERVISOR>    Supervisor_history;
};
```

Figure 26.10
Possible ODL schema for a temporal valid time EMPLOYEE_VT object class using attribute versioning.

26.2.4 Temporal Querying Constructs and the TSQL2 Language

So far, we have discussed how data models may be extended with temporal constructs. Now we give a brief overview of how query operations need to be extended for temporal querying. We will briefly discuss the TSQL2 language, which extends SQL for querying valid time, transaction time, and bitemporal relational databases.

In nontemporal relational databases, the typical selection conditions involve attribute conditions, and tuples that satisfy these conditions are selected from the set of

current tuples. Following that, the attributes of interest to the query are specified by a *projection operation* (see Chapter 6). For example, in the query to retrieve the names of all employees working in department 5 whose salary is greater than 30000, the selection condition would be as follows:

((Salary > 30000) AND (Dno = 5))

The projected attribute would be Name. In a temporal database, the conditions may involve time in addition to attributes. A **pure time condition** involves only time—for example, to select all employee tuple versions that were valid on a certain *time point T* or that were valid *during a certain time period* $[T_1, T_2]$. In this case, the specified time period is compared with the valid time period of each tuple version [*T*.Vst, *T*.Vet], and only those tuples that satisfy the condition are selected. In these operations, a period is considered to be equivalent to the set of time points from T_1 to T_2 inclusive, so the standard set comparison operations can be used. Additional operations, such as whether one time period ends *before* another starts are also needed.[21]

Some of the more common operations used in queries are as follows:

[*T*.Vst, *T*.Vet] **INCLUDES** $[T_1, T_2]$	Equivalent to $T_1 \geq T$.Vst AND $T_2 \leq T$.Vet
[*T*.Vst, *T*.Vet] **INCLUDED_IN** $[T_1, T_2]$	Equivalent to $T_1 \leq T$.Vst AND $T_2 \geq T$.Vet
[*T*.Vst, *T*.Vet] **OVERLAPS** $[T_1, T_2]$	Equivalent to ($T_1 \leq T$.Vet AND $T_2 \geq T$.Vst)[22]
[*T*.Vst, *T*.Vet] **BEFORE** $[T_1, T_2]$	Equivalent to $T_1 \geq T$.Vet
[*T*.Vst, *T*.Vet] **AFTER** $[T_1, T_2]$	Equivalent to $T_2 \leq T$.Vst
[*T*.Vst, *T*.Vet] **MEETS_BEFORE** $[T_1, T_2]$	Equivalent to $T_1 = T$.Vet + 1[23]
[*T*.Vst, *T*.Vet] **MEETS_AFTER** $[T_1, T_2]$	Equivalent to $T_2 + 1 = T$.Vst

Additionally, operations are needed to manipulate time periods, such as computing the union or intersection of two time periods. The results of these operations may not themselves be periods, but rather **temporal elements**—a collection of one or more *disjoint* time periods such that no two time periods in a temporal element are directly adjacent. That is, for any two time periods $[T_1, T_2]$ and $[T_3, T_4]$ in a temporal element, the following three conditions must hold:

- $[T_1, T_2]$ intersection $[T_3, T_4]$ is empty.
- T_3 is not the time point following T_2 in the given granularity.
- T_1 is not the time point following T_4 in the given granularity.

The latter conditions are necessary to ensure unique representations of temporal elements. If two time periods $[T_1, T_2]$ and $[T_3, T_4]$ are adjacent, they are combined

[21]A complete set of operations, known as **Allen's algebra** (Allen, 1983), has been defined for comparing time periods.

[22]This operation returns true if the *intersection* of the two periods is not empty; it has also been called INTERSECTS_WITH.

[23]Here, 1 refers to one time point in the specified granularity. The MEETS operations basically specify if one period starts immediately after another period ends.

into a single time period [T_1, T_4]. This is called **coalescing** of time periods. Coalescing also combines intersecting time periods.

To illustrate how pure time conditions can be used, suppose a user wants to select all employee versions that were valid at any point during 2002. The appropriate selection condition applied to the relation in Figure 26.8 would be

[*T*.Vst, *T*.Vet] **OVERLAPS** [2002-01-01, 2002-12-31]

Typically, most temporal selections are applied to the valid time dimension. For a bitemporal database, one usually applies the conditions to the currently correct tuples with *uc* as their transaction end times. However, if the query needs to be applied to a previous database state, an AS_OF *T* clause is appended to the query, which means that the query is applied to the valid time tuples that were correct in the database at time *T*.

In addition to pure time conditions, other selections involve **attribute and time conditions**. For example, suppose we wish to retrieve all EMP_VT tuple versions T for employees who worked in department 5 at any time during 2002. In this case, the condition is

[*T*.Vst, *T*.Vet]**OVERLAPS** [2002-01-01, 2002-12-31] AND (*T*.Dno = 5)

Finally, we give a brief overview of the TSQL2 query language, which extends SQL with constructs for temporal databases. The main idea behind TSQL2 is to allow users to specify whether a relation is nontemporal (that is, a standard SQL relation) or temporal. The CREATE TABLE statement is extended with an *optional* AS clause to allow users to declare different temporal options. The following options are available:

- <AS VALID STATE <GRANULARITY> (valid time relation with valid time period)
- <AS VALID EVENT <GRANULARITY> (valid time relation with valid time point)
- <AS TRANSACTION (transaction time relation with transaction time period)
- <AS VALID STATE <GRANULARITY> AND TRANSACTION (bitemporal relation, valid time period)
- <AS VALID EVENT <GRANULARITY> AND TRANSACTION (bitemporal relation, valid time point)

The keywords STATE and EVENT are used to specify whether a time *period* or time *point* is associated with the valid time dimension. In TSQL2, rather than have the user actually see how the temporal tables are implemented (as we discussed in the previous sections), the TSQL2 language adds query language constructs to specify various types of temporal selections, temporal projections, temporal aggregations, transformation among granularities, and many other concepts. The book by Snodgrass et al. (1995) describes the language.

26.2.5 Time Series Data

Time series data is used very often in financial, sales, and economics applications. They involve data values that are recorded according to a specific predefined sequence of time points. Therefore, they are a special type of **valid event data**, where the event time points are predetermined according to a fixed calendar. Consider the example of closing daily stock prices of a particular company on the New York Stock Exchange. The granularity here is day, but the days that the stock market is open are known (nonholiday weekdays). Hence, it has been common to specify a computational procedure that calculates the particular **calendar** associated with a time series. Typical queries on time series involve **temporal aggregation** over higher granularity intervals—for example, finding the average or maximum *weekly* closing stock price or the maximum and minimum *monthly* closing stock price from the *daily* information.

As another example, consider the daily sales dollar amount at each store of a chain of stores owned by a particular company. Again, typical temporal aggregates would be retrieving the weekly, monthly, or yearly sales from the daily sales information (using the sum aggregate function), or comparing same store monthly sales with previous monthly sales, and so on.

Because of the specialized nature of time series data and the lack of support for it in older DBMSs, it has been common to use specialized **time series management systems** rather than general-purpose DBMSs for managing such information. In such systems, it has been common to store time series values in sequential order in a file, and apply specialized time series procedures to analyze the information. The problem with this approach is that the full power of high-level querying in languages such as SQL will not be available in such systems.

More recently, some commercial DBMS packages are offering time series extensions, such as the Oracle time cartridge and the time series data blade of Informix Universal Server. In addition, the TSQL2 language provides some support for time series in the form of event tables.

26.3 Spatial Database Concepts[24]

26.3.1 Introduction to Spatial Databases

Spatial databases incorporate functionality that provides support for databases that keep track of objects in a multidimensional space. For example, cartographic databases that store maps include two-dimensional spatial descriptions of their objects—from countries and states to rivers, cities, roads, seas, and so on. The systems that manage geographic data and related applications are known as

[24]The contribution of Pranesh Parimala Ranganathan to this section is appreciated.

Geographical Information Systems (**GIS**), and they are used in areas such as environmental applications, transportation systems, emergency response systems, and battle management. Other databases, such as meteorological databases for weather information, are three-dimensional, since temperatures and other meteorological information are related to three-dimensional spatial points. In general, a **spatial database** stores objects that have spatial characteristics that describe them and that have spatial relationships among them. The spatial relationships among the objects are important, and they are often needed when querying the database. Although a spatial database can in general refer to an *n*-dimensional space for any *n,* we will limit our discussion to two dimensions as an illustration.

A spatial database is optimized to store and query data related to objects in space, including points, lines and polygons. Satellite images are a prominent example of spatial data. Queries posed on these spatial data, where predicates for selection deal with spatial parameters, are called **spatial queries**. For example, "What are the names of all bookstores within five miles of the College of Computing building at Georgia Tech?" is a spatial query. Whereas typical databases process numeric and character data, additional functionality needs to be added for databases to process spatial data types. A query such as "List all the customers located within twenty miles of company headquarters" will require the processing of spatial data types typically outside the scope of standard relational algebra and may involve consulting an external geographic database that maps the company headquarters and each customer to a 2-D map based on their address. Effectively, each customer will be associated to a <latitude, longitude> position. A traditional B^+-tree index based on customers' zip codes or other nonspatial attributes cannot be used to process this query since traditional indexes are not capable of ordering multidimensional coordinate data. Therefore, there is a special need for databases tailored for handling spatial data and spatial queries.

Table 26.1 shows the common analytical operations involved in processing geographic or spatial data.[25] **Measurement operations** are used to measure some

Table 26.1 Common Types of Analysis for Spatial Data

Analysis Type	Type of Operations and Measurements
Measurements	Distance, perimeter, shape, adjacency, and direction
Spatial analysis/statistics	Pattern, autocorrelation, and indexes of similarity and topology using spatial and nonspatial data
Flow analysis	Connectivity and shortest path
Location analysis	Analysis of points and lines within a polygon
Terrain analysis	Slope/aspect, catchment area, drainage network
Search	Thematic search, search by region

[25]List of GIS analysis operations as proposed in Albrecht (1996).

global properties of single objects (such as the area, the relative size of an object's parts, compactness, or symmetry), and to measure the relative position of different objects in terms of distance and direction. **Spatial analysis** operations, which often use statistical techniques, are used to uncover *spatial relationships* within and among mapped data layers. An example would be to create a map—known as a *prediction map*—that identifies the locations of likely customers for particular products based on the historical sales and demographic information. **Flow analysis** operations help in determining the shortest path between two points and also the connectivity among nodes or regions in a graph. **Location analysis** aims to find if the given set of points and lines lie within a given polygon (location). The process involves generating a buffer around existing geographic features and then identifying or selecting features based on whether they fall inside or outside the boundary of the buffer. **Digital terrain analysis** is used to build three-dimensional models, where the topography of a geographical location can be represented with an x, y, z data model known as Digital Terrain (or Elevation) Model (DTM/DEM). The x and y dimensions of a DTM represent the horizontal plane, and z represents spot heights for the respective x, y coordinates. Such models can be used for analysis of environmental data or during the design of engineering projects that require terrain information. Spatial search allows a user to search for objects within a particular spatial region. For example, **thematic search** allows us to search for objects related to a particular theme or class, such as "Find all water bodies within 25 miles of Atlanta" where the class is *water*.

There are also **topological relationships** among spatial objects. These are often used in Boolean predicates to select objects based on their spatial relationships. For example, if a city boundary is represented as a polygon and freeways are represented as multilines, a condition such as "Find all freeways that go through Arlington, Texas" would involve an *intersects* operation, to determine which freeways (lines) intersect the city boundary (polygon).

26.3.2 Spatial Data Types and Models

This section briefly describes the common data types and models for storing spatial data. Spatial data comes in three basic forms. These forms have become a *de facto* standard due to their wide use in commercial systems.

- **Map Data**[26] includes various geographic or spatial features of objects in a map, such as an object's shape and the location of the object within the map. The three basic types of features are points, lines, and polygons (or areas). **Points** are used to represent spatial characteristics of objects whose locations correspond to a single 2-d coordinate (x, y, or longitude/latitude) in the scale of a particular application. Depending on the scale, some examples of point objects could be buildings, cellular towers, or stationary vehicles. Moving

[26]These types of geographic data are based on ESRI's guide to GIS. See www.gis.com/implementing_gis/data/data_types.html

vehicles and other moving objects can be represented by a sequence of point locations that change over time. **Lines** represent objects having length, such as roads or rivers, whose spatial characteristics can be approximated by a sequence of connected lines. **Polygons** are used to represent spatial characteristics of objects that have a boundary, such as countries, states, lakes, or cities. Notice that some objects, such as buildings or cities, can be represented as either points or polygons, depending on the scale of detail.

- **Attribute data** is the descriptive data that GIS systems associate with **map features**. For example, suppose that a map contains features that represent counties within a US state (such as Texas or Oregon). Attributes for each county feature (object) could include population, largest city/town, area in square miles, and so on. Other attribute data could be included for other features in the map, such as states, cities, congressional districts, census tracts, and so on.
- **Image data** includes data such as satellite images and aerial photographs, which are typically created by cameras. Objects of interest, such as buildings and roads, can be identified and overlaid on these images. Images can also be attributes of map features. One can add images to other map features so that clicking on the feature would display the image. Aerial and satellite images are typical examples of raster data.

Models of spatial information are sometimes grouped into two broad categories: *field* and *object*. A spatial application (such as remote sensing or highway traffic control) is modeled using either a field- or an object-based model, depending on the requirements and the traditional choice of model for the application. **Field models** are often used to model spatial data that is continuous in nature, such as terrain elevation, temperature data, and soil variation characteristics, whereas **object models** have traditionally been used for applications such as transportation networks, land parcels, buildings, and other objects that possess both spatial and non-spatial attributes.

26.3.3 Spatial Operators

Spatial operators are used to capture all the relevant geometric properties of objects embedded in the physical space and the relations between them, as well as to perform spatial analysis. Operators are classified into three broad categories.

- **Topological operators.** Topological properties are invariant when topological transformations are applied. These properties do not change after transformations like rotation, translation, or scaling. Topological operators are hierarchically structured in several levels, where the base level offers operators the ability to check for detailed topological relations between regions with a broad boundary, and the higher levels offer more abstract operators that allow users to query uncertain spatial data independent of the underlying geometric data model. Examples include open (region), close (region), and inside (point, loop).

- **Projective operators.** Projective operators, such as *convex hull*, are used to express predicates about the concavity/convexity of objects as well as other spatial relations (for example, being inside the concavity of a given object).
- **Metric operators.** Metric operators provide a more specific description of the object's geometry. They are used to measure some global properties of single objects (such as the area, relative size of an object's parts, compactness, and symmetry), and to measure the relative position of different objects in terms of distance and direction. Examples include length (arc) and distance (point, point).

Dynamic Spatial Operators. The operations performed by the operators mentioned above are static, in the sense that the operands are not affected by the application of the operation. For example, calculating the length of the curve has no effect on the curve itself. **Dynamic operations** alter the objects upon which the operations act. The three fundamental dynamic operations are *create*, *destroy*, and *update*. A representative example of dynamic operations would be updating a spatial object that can be subdivided into translate (shift position), rotate (change orientation), scale up or down, reflect (produce a mirror image), and shear (deform).

Spatial Queries. Spatial queries are requests for spatial data that require the use of spatial operations. The following categories illustrate three typical types of spatial queries:

- **Range query.** Finds the objects of a particular type that are within a given spatial area or within a particular distance from a given location. (For example, find all hospitals within the Metropolitan Atlanta city area, or find all ambulances within five miles of an accident location.)
- **Nearest neighbor query.** Finds an object of a particular type that is closest to a given location. (For example, find the police car that is closest to the location of crime.)
- **Spatial joins or overlays.** Typically joins the objects of two types based on some spatial condition, such as the objects intersecting or overlapping spatially or being within a certain distance of one another. (For example, find all townships located on a major highway between two cities or find all homes that are within two miles of a lake.)

26.3.4 Spatial Data Indexing

A spatial index is used to organize objects into a set of buckets (which correspond to pages of secondary memory), so that objects in a particular spatial region can be easily located. Each bucket has a bucket region, a part of space containing all objects stored in the bucket. The bucket regions are usually rectangles; for point data structures, these regions are disjoint and they partition the space so that each point belongs to precisely one bucket. There are essentially two ways of providing a spatial index.

1. Specialized indexing structures that allow efficient search for data objects based on spatial search operations are included in the database system. These indexing structures would play a similar role to that performed by B^+-tree indexes in traditional database systems. Examples of these indexing structures are *grid files* and *R-trees*. Special types of spatial indexes, known as *spatial join indexes*, can be used to speed up spatial join operations.
2. Instead of creating brand new indexing structures, the two-dimensional (2-d) spatial data is converted to single-dimensional (1-d) data, so that traditional indexing techniques (B^+-tree) can be used. The algorithms for converting from 2-d to 1-d are known as *space filling curves*. We will not discuss these methods in detail (see the Selected Bibliography for further references).

We give an overview of some of the spatial indexing techniques next.

Grid Files. We introduced grid files for indexing of data on multiple attributes in Chapter 18. They can also be used for indexing 2-dimensional and higher *n*-dimensional spatial data. **The fixed-grid** method divides an *n*-dimensional hyperspace into equal size buckets. The data structure that implements the fixed grid is an *n*-dimensional array. The objects whose spatial locations lie within a cell (totally or partially) can be stored in a dynamic structure to handle overflows. This structure is useful for uniformly distributed data like satellite imagery. However, the fixed-grid structure is rigid, and its directory can be sparse and large.

R-Trees. The **R-tree** is a height-balanced tree, which is an extension of the B^+-tree for *k*-dimensions, where $k > 1$. For two dimensions (2-d), spatial objects are approximated in the R-tree by their **minimum bounding rectangle** (**MBR**), which is the smallest rectangle, with sides parallel to the coordinate system (*x* and *y*) axis, that contains the object. R-trees are characterized by the following properties, which are similar to the properties for B^+-trees (see Section 18.3) but are adapted to 2-d spatial objects. As in Section 18.3, we use *M* to indicate the maximum number of entries that can fit in an R-tree node.

1. The structure of each index entry (or index record) in a leaf node is (I, *object-identifier*), where I is the MBR for the spatial object whose identifier is *object-identifier*.
2. Every node except the root node must be at least half full. Thus, a leaf node that is not the root should contain *m* entries (I, *object-identifier*) where $M/2 <= m <= M$. Similarly, a non-leaf node that is not the root should contain *m* entries (I, *child-pointer*) where $M/2 <= m <= M$, and I is the MBR that contains the union of all the rectangles in the node pointed at by *child-pointer*.
3. All leaf nodes are at the same level, and the root node should have at least two pointers unless it is a leaf node.
4. All MBRs have their sides parallel to the axes of the global coordinate system.

Other spatial storage structures include quadtrees and their variations. **Quadtrees** generally divide each space or subspace into equally sized areas, and proceed with the subdivisions of each subspace to identify the positions of various objects. Recently, many newer spatial access structures have been proposed, and this area remains an active research area.

Spatial Join Index. A spatial join index precomputes a spatial join operation and stores the pointers to the related object in an index structure. Join indexes improve the performance of recurring join queries over tables that have low update rates. Spatial join conditions are used to answer queries such as "Create a list of highway-river combinations that cross." The spatial join is used to identify and retrieve these pairs of objects that satisfy the *cross* spatial relationship. Because computing the results of spatial relationships is generally time consuming, the result can be computed once and stored in a table that has the pairs of object identifiers (or tuple ids) that satisfy the spatial relationship, which is essentially the join index.

A join index can be described by a bipartite graph G = (V1,V2,E), where V1 contains the tuple ids of relation *R*, and V2 contains the tuple ids of relation *S*. Edge set contains an edge (vr,vs) for vr in *R* and vs in *S*, if there is a tuple corresponding to (vr,vs) in the join index. The bipartite graph models all of the related tuples as connected vertices in the graphs. Spatial join indexes are used in operations (see Section 26.3.3) that involve computation of relationships among spatial objects.

26.3.5 Spatial Data Mining

Spatial data tends to be highly correlated. For example, people with similar characteristics, occupations, and backgrounds tend to cluster together in the same neighborhoods.

The three major spatial data mining techniques are spatial classification, spatial association, and spatial clustering.

- **Spatial classification.** The goal of classification is to estimate the value of an attribute of a relation based on the value of the relation's other attributes. An example of the spatial classification problem is determining the locations of nests in a wetland based on the value of other attributes (for example, vegetation durability and water depth); it is also called the *location prediction problem*. Similarly, where to expect hotspots in crime activity is also a location prediction problem.
- **Spatial association. Spatial association rules** are defined in terms of spatial predicates rather than items. A spatial association rule is of the form

  $P_1 \wedge P_2 \wedge \ldots \wedge P_n \Rightarrow Q_1 \wedge Q_2 \wedge \ldots \wedge Q_m$,

  where at least one of the P_i's or Q_j's is a spatial predicate. For example, the rule

  is_a(x, country) ^ touches(x, Mediterranean) ⇒ is_a (x, wine-exporter)

(that is, a country that is adjacent to the Mediterranean Sea is typically a wine exporter) is an example of an association rule, which will have a certain support *s* and confidence *c*.[27]

Spatial colocation rules attempt to generalize association rules to point to collection data sets that are indexed by space. There are several crucial differences between spatial and nonspatial associations including:

1. The notion of a transaction is absent in spatial situations, since data is embedded in continuous space. Partitioning space into transactions would lead to an overestimate or an underestimate of interest measures, for example, support or confidence.
2. Size of item sets in spatial databases is small, that is, there are many fewer items in the item set in a spatial situation than in a nonspatial situation.

In most instances, spatial items are a discrete version of continuous variables. For example, in the United States income regions may be defined as regions where the mean yearly income is within certain ranges, such as, below $40,000, from $40,000 to $100,000, and above $100,000.

- **Spatial Clustering** attempts to group database objects so that the most similar objects are in the same cluster, and objects in different clusters are as dissimilar as possible. One application of spatial clustering is to group together seismic events in order to determine earthquake faults. An example of a spatial clustering algorithm is **density-based clustering**, which tries to find clusters based on the density of data points in a region. These algorithms treat clusters as dense regions of objects in the data space. Two variations of these algorithms are density-based spatial clustering of applications with noise (DBSCAN)[28] and density-based clustering (DENCLUE).[29] DBSCAN is a density-based clustering algorithm because it finds a number of clusters starting from the estimated density distribution of corresponding nodes.

26.3.6 Applications of Spatial Data

Spatial data management is useful in many disciplines, including geography, remote sensing, urban planning, and natural resource management. Spatial database management is playing an important role in the solution of challenging scientific problems such as global climate change and genomics. Due to the spatial nature of genome data, GIS and spatial database management systems have a large role to play in the area of bioinformatics. Some of the typical applications include pattern recognition (for example, to check if the topology of a particular gene in the genome is found in any other sequence feature map in the database), genome

[27]Concepts of support and confidence for association rules are discussed as part of data mining in Section 28.2.

[28]DBSCAN was proposed by Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu (1996).

[29]DENCLUE was proposed by Hinnenberg and Gabriel (2007).

browser development, and visualization maps. Another important application area of spatial data mining is the spatial outlier detection. A **spatial outlier** is a spatially referenced object whose nonspatial attribute values are significantly different from those of other spatially referenced objects in its spatial neighborhood. For example, if a neighborhood of older houses has just one brand-new house, that house would be an outlier based on the nonspatial attribute 'house_age'. Detecting spatial outliers is useful in many applications of geographic information systems and spatial databases. These application domains include transportation, ecology, public safety, public health, climatology, and location-based services.

26.4 Multimedia Database Concepts

Multimedia databases provide features that allow users to store and query different types of multimedia information, which includes *images* (such as photos or drawings), *video clips* (such as movies, newsreels, or home videos), *audio clips* (such as songs, phone messages, or speeches), and *documents* (such as books or articles). The main types of database queries that are needed involve locating multimedia sources that contain certain objects of interest. For example, one may want to locate all video clips in a video database that include a certain person, say Michael Jackson. One may also want to retrieve video clips based on certain activities included in them, such as video clips where a soccer goal is scored by a certain player or team.

The above types of queries are referred to as **content-based retrieval**, because the multimedia source is being retrieved based on its containing certain objects or activities. Hence, a multimedia database must use some model to organize and index the multimedia sources based on their contents. *Identifying the contents* of multimedia sources is a difficult and time-consuming task. There are two main approaches. The first is based on **automatic analysis** of the multimedia sources to identify certain mathematical characteristics of their contents. This approach uses different techniques depending on the type of multimedia source (image, video, audio, or text). The second approach depends on **manual identification** of the objects and activities of interest in each multimedia source and on using this information to index the sources. This approach can be applied to all multimedia sources, but it requires a manual preprocessing phase where a person has to scan each multimedia source to identify and catalog the objects and activities it contains so that they can be used to index the sources.

In the first part of this section, we will briefly discuss some of the characteristics of each type of multimedia source—images, video, audio, and text/documents. Then we will discuss approaches for automatic analysis of images followed by the problem of object recognition in images. We end this section with some remarks on analyzing audio sources.

An **image** is typically stored either in raw form as a set of pixel or cell values, or in compressed form to save space. The image *shape descriptor* describes the geometric shape of the raw image, which is typically a rectangle of **cells** of a certain width and height. Hence, each image can be represented by an m by n grid of cells. Each cell

contains a pixel value that describes the cell content. In black-and-white images, pixels can be one bit. In gray scale or color images, a pixel is multiple bits. Because images may require large amounts of space, they are often stored in compressed form. Compression standards, such as GIF, JPEG, or MPEG, use various mathematical transformations to reduce the number of cells stored but still maintain the main image characteristics. Applicable mathematical transforms include Discrete Fourier Transform (DFT), Discrete Cosine Transform (DCT), and wavelet transforms.

To identify objects of interest in an image, the image is typically divided into homogeneous segments using a *homogeneity predicate.* For example, in a color image, adjacent cells that have similar pixel values are grouped into a segment. The homogeneity predicate defines conditions for automatically grouping those cells. Segmentation and compression can hence identify the main characteristics of an image.

A typical image database query would be to find images in the database that are similar to a given image. The given image could be an isolated segment that contains, say, a pattern of interest, and the query is to locate other images that contain that same pattern. There are two main techniques for this type of search. The first approach uses a **distance function** to compare the given image with the stored images and their segments. If the distance value returned is small, the probability of a match is high. Indexes can be created to group stored images that are close in the distance metric so as to limit the search space. The second approach, called the **transformation approach**, measures image similarity by having a small number of transformations that can change one image's cells to match the other image. Transformations include rotations, translations, and scaling. Although the transformation approach is more general, it is also more time-consuming and difficult.

A **video source** is typically represented as a sequence of frames, where each frame is a still image. However, rather than identifying the objects and activities in every individual frame, the video is divided into **video segments**, where each segment comprises a sequence of contiguous frames that includes the same objects/activities. Each segment is identified by its starting and ending frames. The objects and activities identified in each video segment can be used to index the segments. An indexing technique called *frame segment trees* has been proposed for video indexing. The index includes both objects, such as persons, houses, and cars, as well as activities, such as a person *delivering* a speech or two people *talking*. Videos are also often compressed using standards such as MPEG.

Audio sources include stored recorded messages, such as speeches, class presentations, or even surveillance recordings of phone messages or conversations by law enforcement. Here, discrete transforms can be used to identify the main characteristics of a certain person's voice in order to have similarity-based indexing and retrieval. We will briefly comment on their analysis in Section 26.4.4.

A **text/document source** is basically the full text of some article, book, or magazine. These sources are typically indexed by identifying the keywords that appear in the text and their relative frequencies. However, filler words or common words called **stopwords** are eliminated from the process. Because there can be many keywords

when attempting to index a collection of documents, techniques have been developed to reduce the number of keywords to those that are most relevant to the collection. A dimensionality reduction technique called *singular value decompositions* (SVD), which is based on matrix transformations, can be used for this purpose. An indexing technique called *telescoping vector trees* (TV-trees), can then be used to group similar documents. Chapter 27 discusses document processing in detail.

26.4.1 Automatic Analysis of Images

Analysis of multimedia sources is critical to support any type of query or search interface. We need to represent multimedia source data such as images in terms of features that would enable us to define similarity. The work done so far in this area uses low-level visual features such as color, texture, and shape, which are directly related to the perceptual aspects of image content. These features are easy to extract and represent, and it is convenient to design similarity measures based on their statistical properties.

Color is one of the most widely used visual features in content-based image retrieval since it does not depend upon image size or orientation. Retrieval based on color similarity is mainly done by computing a color histogram for each image that identifies the proportion of pixels within an image for the three color channels (red, green, blue—**RGB**). However, RGB representation is affected by the orientation of the object with respect to illumination and camera direction. Therefore, current image retrieval techniques compute color histograms using competing invariant representations such as **HSV** (hue, saturation, value). HSV describes colors as points in a cylinder whose central axis ranges from black at the bottom to white at the top with neutral colors between them. The angle around the axis corresponds to the hue, the distance from the axis corresponds to the saturation, and the distance along the axis corresponds to the value (brightness).

Texture refers to the patterns in an image that present the properties of homogeneity that do not result from the presence of a single color or intensity value. Examples of texture classes are rough and silky. Examples of textures that can be identified include pressed calf leather, straw matting, cotton canvas, and so on. Just as pictures are represented by arrays of pixels (picture elements), textures are represented by **arrays of texels** (texture elements). These textures are then placed into a number of sets, depending on how many textures are identified in the image. These sets not only contain the texture definition but also indicate where in the image the texture is located. Texture identification is primarily done by modeling it as a two-dimensional, gray-level variation. The relative brightness of pairs of pixels is computed to estimate the degree of contrast, regularity, coarseness, and directionality.

Shape refers to the shape of a region within an image. It is generally determined by applying segmentation or edge detection to an image. **Segmentation** is a region-based approach that uses an entire region (sets of pixels), whereas **edge detection** is a boundary-based approach that uses only the outer boundary characteristics of entities. Shape representation is typically required to be invariant to translation,

rotation, and scaling. Some well-known methods for shape representation include Fourier descriptors and moment invariants.

26.4.2 Object Recognition in Images

Object recognition is the task of identifying real-world objects in an image or a video sequence. The system must be able to identify the object even when the images of the object vary in viewpoints, size, scale, or even when they are rotated or translated. Some approaches have been developed to divide the original image into regions based on similarity of contiguous pixels. Thus, in a given image showing a tiger in the jungle, a tiger subimage may be detected against the background of the jungle, and when compared with a set of training images, it may be tagged as a tiger.

The representation of the multimedia object in an object model is extremely important. One approach is to divide the image into homogeneous segments using a homogeneous predicate. For example, in a colored image, adjacent cells that have similar pixel values are grouped into a segment. The homogeneity predicate defines conditions for automatically grouping those cells. Segmentation and compression can hence identify the main characteristics of an image. Another approach finds measurements of the object that are invariant to transformations. It is impossible to keep a database of examples of all the different transformations of an image. To deal with this, object recognition approaches find interesting points (or features) in an image that are invariant to transformations.

An important contribution to this field was made by Lowe,[30] who used scale-invariant features from images to perform reliable object recognition. This approach is called **scale-invariant feature transform** (**SIFT**). The SIFT features are invariant to image scaling and rotation, and partially invariant to change in illumination and 3D camera viewpoint. They are well localized in both the spatial and frequency domains, reducing the probability of disruption by occlusion, clutter, or noise. In addition, the features are highly distinctive, which allows a single feature to be correctly matched with high probability against a large database of features, providing a basis for object and scene recognition.

For image matching and recognition, SIFT features (also known as *keypoint features*) are first extracted from a set of reference images and stored in a database. Object recognition is then performed by comparing each feature from the new image with the features stored in the database and finding candidate matching features based on the Euclidean distance of their feature vectors. Since the keypoint features are highly distinctive, a single feature can be correctly matched with good probability in a large database of features.

In addition to SIFT, there are a number of competing methods available for object recognition under clutter or partial occlusion. For example, **RIFT**, a rotation invariant generalization of SIFT, identifies groups of local affine regions (image features

[30]See Lowe (2004), "Distinctive Image Features from Scale-Invariant Keypoints."

having a characteristic appearance and elliptical shape) that remain approximately affinely rigid across a range of views of an object, and across multiple instances of the same object class.

26.4.3 Semantic Tagging of Images

The notion of implicit tagging is an important one for image recognition and comparison. Multiple tags may attach to an image or a subimage: for instance, in the example we referred to above, tags such as "tiger," "jungle," "green," and "stripes" may be associated with that image. Most image search techniques retrieve images based on user-supplied tags that are often not very accurate or comprehensive. To improve search quality, a number of recent systems aim at automated generation of these image tags. In case of multimedia data, most of its semantics is present in its content. These systems use image-processing and statistical-modeling techniques to analyze image content to generate accurate annotation tags that can then be used to retrieve images by content. Since different annotation schemes will use different vocabularies to annotate images, the quality of image retrieval will be poor. To solve this problem, recent research techniques have proposed the use of concept hierarchies, taxonomies, or ontologies using **OWL** (**Web Ontology Language**), in which terms and their relationships are clearly defined. These can be used to infer higher-level concepts based on tags. Concepts like "sky" and "grass" may be further divided into "clear sky" and "cloudy sky" or "dry grass" and "green grass" in such a taxonomy. These approaches generally come under semantic tagging and can be used in conjunction with the above feature-analysis and object-identification strategies.

26.4.4 Analysis of Audio Data Sources

Audio sources are broadly classified into speech, music, and other audio data. Each of these are significantly different from the other, hence different types of audio data are treated differently. Audio data must be digitized before it can be processed and stored. Indexing and retrieval of audio data is arguably the toughest among all types of media, because like video, it is continuous in time and does not have easily measurable characteristics such as text. Clarity of sound recordings is easy to perceive humanly but is hard to quantify for machine learning. Interestingly, speech data often uses speech recognition techniques to aid the actual audio content, as this can make indexing this data a lot easier and more accurate. This is sometimes referred to as *text-based indexing of audio data*. The speech metadata is typically content dependent, in that the metadata is generated from the audio content, for example, the length of the speech, the number of speakers, and so on. However, some of the metadata might be independent of the actual content, such as the length of the speech and the format in which the data is stored. Music indexing, on the other hand, is done based on the statistical analysis of the audio signal, also known as *content-based indexing*. Content-based indexing often makes use of the key features of sound: intensity, pitch, timbre, and rhythm. It is possible to compare different pieces of audio data and retrieve information from them based on the calculation of certain features, as well as application of certain transforms.

26.5 Introduction to Deductive Databases

26.5.1 Overview of Deductive Databases

In a deductive database system we typically specify rules through a **declarative language**—a language in which we specify what to achieve rather than how to achieve it. An **inference engine** (or **deduction mechanism**) within the system can deduce new facts from the database by interpreting these rules. The model used for deductive databases is closely related to the relational data model, and particularly to the domain relational calculus formalism (see Section 6.6). It is also related to the field of **logic programming** and the **Prolog** language. The deductive database work based on logic has used Prolog as a starting point. A variation of Prolog called **Datalog** is used to define rules declaratively in conjunction with an existing set of relations, which are themselves treated as literals in the language. Although the language structure of Datalog resembles that of Prolog, its operational semantics—that is, how a Datalog program is executed—is still different.

A deductive database uses two main types of specifications: facts and rules. **Facts** are specified in a manner similar to the way relations are specified, except that it is not necessary to include the attribute names. Recall that a tuple in a relation describes some real-world fact whose meaning is partly determined by the attribute names. In a deductive database, the meaning of an attribute value in a tuple is determined solely by its *position* within the tuple. **Rules** are somewhat similar to relational views. They specify virtual relations that are not actually stored but that can be formed from the facts by applying inference mechanisms based on the rule specifications. The main difference between rules and views is that rules may involve recursion and hence may yield virtual relations that cannot be defined in terms of basic relational views.

The evaluation of Prolog programs is based on a technique called *backward chaining,* which involves a top-down evaluation of goals. In the deductive databases that use Datalog, attention has been devoted to handling large volumes of data stored in a relational database. Hence, evaluation techniques have been devised that resemble those for a bottom-up evaluation. Prolog suffers from the limitation that the order of specification of facts and rules is significant in evaluation; moreover, the order of literals (defined in Section 26.5.3) within a rule is significant. The execution techniques for Datalog programs attempt to circumvent these problems.

26.5.2 Prolog/Datalog Notation

The notation used in Prolog/Datalog is based on providing predicates with unique names. A **predicate** has an implicit meaning, which is suggested by the predicate name, and a fixed number of **arguments**. If the arguments are all constant values, the predicate simply states that a certain fact is true. If, on the other hand, the predicate has variables as arguments, it is either considered as a query or as part of a rule or constraint. In our discussion, we adopt the Prolog convention that all **constant**

(a)

Facts

SUPERVISE(franklin, john).
SUPERVISE(franklin, ramesh).
SUPERVISE(franklin, joyce).
SUPERVISE(jennifer, alicia).
SUPERVISE(jennifer, ahmad).
SUPERVISE(james, franklin).
SUPERVISE(james, jennifer).
. . .

Rules

SUPERIOR(*X, Y*) :– SUPERVISE(*X, Y*).
SUPERIOR(*X, Y*) :– SUPERVISE(*X, Z*), SUPERIOR(*Z, Y*).
SUBORDINATE(*X, Y*) :– SUPERIOR(*Y, X*).

Queries

SUPERIOR(james, *Y*)?
SUPERIOR(james, joyce)?

(b)

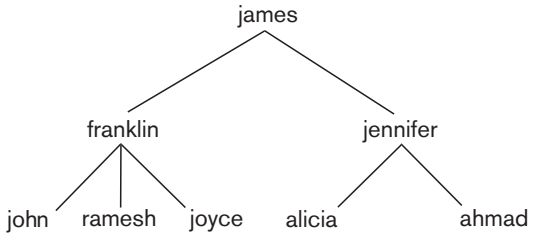


Figure 26.11
(a) Prolog notation.
(b) The supervisory tree.

values in a predicate are either *numeric* or *character strings*; they are represented as identifiers (or names) that start with a *lowercase letter*, whereas **variable names** always start with an *uppercase letter.*

Consider the example shown in Figure 26.11, which is based on the relational database in Figure 3.6, but in a much simplified form. There are three predicate names: *supervise, superior,* and *subordinate.* The SUPERVISE predicate is defined via a set of facts, each of which has two arguments: a supervisor name, followed by the name of a *direct* supervisee (subordinate) of that supervisor. These facts correspond to the actual data that is stored in the database, and they can be considered as constituting a set of tuples in a relation SUPERVISE with two attributes whose schema is

SUPERVISE(Supervisor, Supervisee)

Thus, SUPERVISE(*X, Y*) states the fact that *X supervises Y.* Notice the omission of the attribute names in the Prolog notation. Attribute names are only represented by virtue of the position of each argument in a predicate: the first argument represents the supervisor, and the second argument represents a direct subordinate.

The other two predicate names are defined by rules. The main contributions of deductive databases are the ability to specify recursive rules and to provide a framework for inferring new information based on the specified rules. A rule is of the form **head :– body**, where :– is read as *if and only if.* A rule usually has a **single predicate** to the left of the :– symbol—called the **head** or **left-hand side** (LHS) or **conclusion** of the rule—and **one or more predicates** to the right of the :– symbol—called the **body** or **right-hand side** (RHS) or **premise(s)** of the rule. A predicate with constants as arguments is said to be **ground**; we also refer to it as an **instantiated predicate**. The arguments of the predicates that appear in a rule typically include a number of variable symbols, although predicates can also contain

constants as arguments. A rule specifies that, if a particular assignment or **binding** of constant values to the variables in the body (RHS predicates) makes *all* the RHS predicates **true**, it also makes the head (LHS predicate) true by using the same assignment of constant values to variables. Hence, a rule provides us with a way of generating new facts that are instantiations of the head of the rule. These new facts are based on facts that already exist, corresponding to the instantiations (or bindings) of predicates in the body of the rule. Notice that by listing multiple predicates in the body of a rule we implicitly apply the **logical AND** operator to these predicates. Hence, the commas between the RHS predicates may be read as meaning *and*.

Consider the definition of the predicate SUPERIOR in Figure 26.11, whose first argument is an employee name and whose second argument is an employee who is either a *direct* or an *indirect* subordinate of the first employee. By *indirect subordinate,* we mean the subordinate of some subordinate down to any number of levels. Thus SUPERIOR(X, Y) stands for the fact that *X is a superior of Y* through direct or indirect supervision. We can write two rules that together specify the meaning of the new predicate. The first rule under Rules in the figure states that for every value of X and Y, if SUPERVISE(X, Y)—the rule body—is true, then SUPERIOR(X, Y)—the rule head—is also true, since Y would be a direct subordinate of X (at one level down). This rule can be used to generate all direct superior/subordinate relationships from the facts that define the SUPERVISE predicate. The second recursive rule states that if SUPERVISE(X, Z) *and* SUPERIOR(Z, Y) are *both* true, then SUPERIOR(X, Y) is also true. This is an example of a **recursive rule**, where one of the rule body predicates in the RHS is the same as the rule head predicate in the LHS. In general, the rule body defines a number of premises such that if they are all true, we can deduce that the conclusion in the rule head is also true. Notice that if we have two (or more) rules with the same head (LHS predicate), it is equivalent to saying that the predicate is true (that is, that it can be instantiated) if *either one* of the bodies is true; hence, it is equivalent to a **logical OR** operation. For example, if we have two rules X :– Y and X :– Z, they are equivalent to a rule X :– Y OR Z. The latter form is not used in deductive systems, however, because it is not in the standard form of rule, called a *Horn clause*, as we discuss in Section 26.5.4.

A Prolog system contains a number of **built-in** predicates that the system can interpret directly. These typically include the equality comparison operator =(X, Y), which returns true if X and Y are identical and can also be written as X=Y by using the standard infix notation.[31] Other comparison operators for numbers, such as <, <=, >, and >=, can be treated as binary predicates. Arithmetic functions such as +, –, *, and / can be used as arguments in predicates in Prolog. In contrast, Datalog (in its basic form) does *not* allow functions such as arithmetic operations as arguments; indeed, this is one of the main differences between Prolog and Datalog. However, extensions to Datalog have been proposed that do include functions.

[31]A Prolog system typically has a number of different equality predicates that have different interpretations.

A **query** typically involves a predicate symbol with some variable arguments, and its meaning (or *answer*) is to deduce all the different constant combinations that, when **bound** (assigned) to the variables, can make the predicate true. For example, the first query in Figure 26.11 requests the names of all subordinates of *james* at any level. A different type of query, which has only constant symbols as arguments, returns either a true or a false result, depending on whether the arguments provided can be deduced from the facts and rules. For example, the second query in Figure 26.11 returns true, since SUPERIOR(james, joyce) can be deduced.

26.5.3 Datalog Notation

In Datalog, as in other logic-based languages, a program is built from basic objects called **atomic formulas**. It is customary to define the syntax of logic-based languages by describing the syntax of atomic formulas and identifying how they can be combined to form a program. In Datalog, atomic formulas are **literals** of the form $p(a_1, a_2, ..., a_n)$, where p is the predicate name and n is the number of arguments for predicate p. Different predicate symbols can have different numbers of arguments, and the number of arguments n of predicate p is sometimes called the **arity** or **degree** of p. The arguments can be either constant values or variable names. As mentioned earlier, we use the convention that constant values either are numeric or start with a *lowercase* character, whereas variable names always start with an *uppercase* character.

A number of **built-in predicates** are included in Datalog, which can also be used to construct atomic formulas. The built-in predicates are of two main types: the binary comparison predicates < (less), <= (less_or_equal), > (greater), and >= (greater_or_equal) over ordered domains; and the comparison predicates = (equal) and /= (not_equal) over ordered or unordered domains. These can be used as binary predicates with the same functional syntax as other predicates—for example, by writing less(X, 3)—or they can be specified by using the customary infix notation $X<3$. Note that because the domains of these predicates are potentially infinite, they should be used with care in rule definitions. For example, the predicate greater(X, 3), if used alone, generates an infinite set of values for X that satisfy the predicate (all integer numbers greater than 3).

A **literal** is either an atomic formula as defined earlier—called a **positive literal**—or an atomic formula preceded by **not**. The latter is a negated atomic formula, called a **negative literal**. Datalog programs can be considered to be a *subset* of the predicate calculus formulas, which are somewhat similar to the formulas of the domain relational calculus (see Section 6.7). In Datalog, however, these formulas are first converted into what is known as **clausal form** before they are expressed in Datalog, and only formulas given in a restricted clausal form, called *Horn clauses*,[32] can be used in Datalog.

[32]Named after the mathematician Alfred Horn.

26.5.4 Clausal Form and Horn Clauses

Recall from Section 6.6 that a formula in the relational calculus is a condition that includes predicates called *atoms* (based on relation names). Additionally, a formula can have quantifiers—namely, the *universal quantifier* (for all) and the *existential quantifier* (there exists). In clausal form, a formula must be transformed into another formula with the following characteristics:

- All variables in the formula are universally quantified. Hence, it is not necessary to include the universal quantifiers (for all) explicitly; the quantifiers are removed, and all variables in the formula are *implicitly* quantified by the universal quantifier.
- In clausal form, the formula is made up of a number of clauses, where each **clause** is composed of a number of *literals* connected by OR logical connectives only. Hence, each clause is a *disjunction* of literals.
- The *clauses themselves* are connected by AND logical connectives only, to form a formula. Hence, the **clausal form of a formula** is a *conjunction* of clauses.

It can be shown that *any formula can be converted into clausal form.* For our purposes, we are mainly interested in the form of the individual clauses, each of which is a disjunction of literals. Recall that literals can be positive literals or negative literals. Consider a clause of the form:

$$\text{NOT}(P_1) \text{ OR } \text{NOT}(P_2) \text{ OR } \ldots \text{ OR } \text{NOT}(P_n) \text{ OR } Q_1 \text{ OR } Q_2 \text{ OR } \ldots \text{ OR } Q_m \quad (1)$$

This clause has n negative literals and m positive literals. Such a clause can be transformed into the following equivalent logical formula:

$$P_1 \text{ AND } P_2 \text{ AND } \ldots \text{ AND } P_n \Rightarrow Q_1 \text{ OR } Q_2 \text{ OR } \ldots \text{ OR } Q_m \quad (2)$$

where $\Rightarrow$ is the **implies** symbol. The formulas (1) and (2) are equivalent, meaning that their truth values are always the same. This is the case because if all the P_i literals ($i = 1, 2, \ldots, n$) are true, the formula (2) is true only if at least one of the Q_i's is true, which is the meaning of the $\Rightarrow$ (implies) symbol. For formula (1), if all the P_i literals ($i = 1, 2, \ldots, n$) are true, their negations are all false; so in this case formula (1) is true only if at least one of the Q_i's is true. In Datalog, rules are expressed as a restricted form of clauses called **Horn clauses**, in which a clause can contain *at most one* positive literal. Hence, a Horn clause is either of the form

$$\text{NOT } (P_1) \text{ OR } \text{NOT}(P_2) \text{ OR } \ldots \text{ OR } \text{NOT}(P_n) \text{ OR } Q \quad (3)$$

or of the form

$$\text{NOT } (P_1) \text{ OR } \text{NOT}(P_2) \text{ OR } \ldots \text{ OR } \text{NOT}(P_n) \quad (4)$$

The Horn clause in (3) can be transformed into the clause

$$P_1 \text{ AND } P_2 \text{ AND } \ldots \text{ AND } P_n \Rightarrow Q \quad (5)$$

which is written in Datalog as the following rule:

$$Q :- P_1, P_2, \ldots, P_n. \quad (6)$$

The Horn clause in (4) can be transformed into

$$P_1 \text{ AND } P_2 \text{ AND } \ldots \text{ AND } P_n \Rightarrow \quad (7)$$

which is written in Datalog as follows:

$$P_1, P_2, \ldots, P_n. \quad (8)$$

A **Datalog rule**, as in (6), is hence a Horn clause, and its meaning, based on formula (5), is that if the predicates P_1 AND P_2 AND ... AND P_n are all true for a particular binding to their variable arguments, then Q is also true and can hence be inferred. The Datalog expression (8) can be considered as an integrity constraint, where all the predicates must be true to satisfy the query.

In general, a **query in Datalog** consists of two components:

- A Datalog program, which is a finite set of rules
- A literal $P(X_1, X_2, \ldots, X_n)$, where each X_i is a variable or a constant

A Prolog or Datalog system has an internal **inference engine** that can be used to process and compute the results of such queries. Prolog inference engines typically return one result to the query (that is, one set of values for the variables in the query) at a time and must be prompted to return additional results. On the contrary, Datalog returns results set-at-a-time.

26.5.5 Interpretations of Rules

There are two main alternatives for interpreting the theoretical meaning of rules: *proof-theoretic* and *model-theoretic*. In practical systems, the inference mechanism within a system defines the exact interpretation, which may not coincide with either of the two theoretical interpretations. The inference mechanism is a computational procedure and hence provides a computational interpretation of the meaning of rules. In this section, first we discuss the two theoretical interpretations. Then we briefly discuss inference mechanisms as a way of defining the meaning of rules.

In the **proof-theoretic** interpretation of rules, we consider the facts and rules to be true statements, or **axioms**. **Ground axioms** contain no variables. The facts are ground axioms that are given to be true. Rules are called **deductive axioms**, since they can be used to deduce new facts. The deductive axioms can be used to construct proofs that derive new facts from existing facts. For example, Figure 26.12 shows how to prove the fact SUPERIOR(james, ahmad) from the rules and facts

1. SUPERIOR(X, Y) :– SUPERVISE(X, Y).	(rule 1)
2. SUPERIOR(X, Y) :– SUPERVISE(X, Z), SUPERIOR(Z, Y).	(rule 2)
3. SUPERVISE(jennifer, ahmad).	(ground axiom, given)
4. SUPERVISE(james, jennifer).	(ground axiom, given)
5. SUPERIOR(jennifer, ahmad).	(apply rule 1 on 3)
6. SUPERIOR(james, ahmad).	(apply rule 2 on 4 and 5)

Figure 26.12
Proving a new fact.

given in Figure 26.11. The proof-theoretic interpretation gives us a procedural or computational approach for computing an answer to the Datalog query. The process of proving whether a certain fact (theorem) holds is known as **theorem proving**.

The second type of interpretation is called the **model-theoretic** interpretation. Here, given a finite or an infinite domain of constant values,[33] we assign to a predicate every possible combination of values as arguments. We must then determine whether the predicate is true or false. In general, it is sufficient to specify the combinations of arguments that make the predicate true, and to state that all other combinations make the predicate false. If this is done for every predicate, it is called an **interpretation** of the set of predicates. For example, consider the interpretation shown in Figure 26.13 for the predicates SUPERVISE and SUPERIOR. This interpretation assigns a truth value (true or false) to every possible combination of argument values (from a finite domain) for the two predicates.

An interpretation is called a **model** for a *specific set of rules* if those rules are *always true* under that interpretation; that is, for any values assigned to the variables in the rules, the head of the rules is true when we substitute the truth values assigned to the predicates in the body of the rule by that interpretation. Hence, whenever a particular substitution (binding) to the variables in the rules is applied, if all the predicates in the body of a rule are true under the interpretation, the predicate in the head of the rule must also be true. The interpretation shown in Figure 26.13 is a model for the two rules shown, since it can never cause the rules to be violated. Notice that a rule is violated if a particular binding of constants to the variables makes all the predicates in the rule body true but makes the predicate in the rule head false. For example, if SUPERVISE(*a, b*) and SUPERIOR(*b, c*) are both true under some interpretation, but SUPERIOR(*a, c*) is not true, the interpretation cannot be a model for the recursive rule:

SUPERIOR(X, Y) :– SUPERVISE(X, Z), SUPERIOR(Z, Y)

In the model-theoretic approach, the meaning of the rules is established by providing a model for these rules. A model is called a **minimal model** for a set of rules if we cannot change any fact from true to false and still get a model for these rules. For example, consider the interpretation in Figure 26.13, and assume that the SUPERVISE predicate is defined by a set of known facts, whereas the SUPERIOR predicate is defined as an interpretation (model) for the rules. Suppose that we add the predicate SUPERIOR(james, bob) to the true predicates. This remains a model for the rules shown, but it is not a minimal model, since changing the truth value of SUPERIOR(james,bob) from true to false still provides us with a model for the rules. The model shown in Figure 26.13 is the minimal model for the set of facts that are defined by the SUPERVISE predicate.

In general, the minimal model that corresponds to a given set of facts in the model-theoretic interpretation should be the same as the facts generated by the proof-

[33]The most commonly chosen domain is finite and is called the *Herbrand Universe.*

Rules
SUPERIOR(*X*, *Y*) :– SUPERVISE(*X*, *Y*).
SUPERIOR(*X*, *Y*) :– SUPERVISE(*X*, *Z*), SUPERIOR(*Z*, *Y*).

Interpretation

Known Facts:
SUPERVISE(franklin, john) is **true**.
SUPERVISE(franklin, ramesh) is **true**.
SUPERVISE(franklin, joyce) is **true**.
SUPERVISE(jennifer, alicia) is **true**.
SUPERVISE(jennifer, ahmad) is **true**.
SUPERVISE(james, franklin) is **true**.
SUPERVISE(james, jennifer) is **true**.
SUPERVISE(*X*, *Y*) is **false** for all other possible (*X*, *Y*) combinations

Derived Facts:
SUPERIOR(franklin, john) is **true**.
SUPERIOR(franklin, ramesh) is **true**.
SUPERIOR(franklin, joyce) is **true**.
SUPERIOR(jennifer, alicia) is **true**.
SUPERIOR(jennifer, ahmad) is **true**.
SUPERIOR(james, franklin) is **true**.
SUPERIOR(james, jennifer) is **true**.
SUPERIOR(james, john) is **true**.
SUPERIOR(james, ramesh) is **true**.
SUPERIOR(james, joyce) is **true**.
SUPERIOR(james, alicia) is **true**.
SUPERIOR(james, ahmad) is **true**.
SUPERIOR(*X*, *Y*) is **false** for all other possible (*X*, *Y*) combinations

Figure 26.13
An interpretation that is a minimal model.

theoretic interpretation for the same original set of ground and deductive axioms. However, this is generally true only for rules with a simple structure. Once we allow negation in the specification of rules, the correspondence between interpretations *does not* hold. In fact, with negation, numerous minimal models are possible for a given set of facts.

A third approach to interpreting the meaning of rules involves defining an inference mechanism that is used by the system to deduce facts from the rules. This inference mechanism would define a **computational interpretation** to the meaning of the rules. The Prolog logic programming language uses its inference mechanism to define the meaning of the rules and facts in a Prolog program. Not all Prolog programs correspond to the proof-theoretic or model-theoretic interpretations; it depends on the type of rules in the program. However, for many simple Prolog programs, the Prolog inference mechanism infers the facts that correspond either to the proof-theoretic interpretation or to a minimal model under the model-theoretic interpretation.

26.5.6 Datalog Programs and Their Safety

There are two main methods of defining the truth values of predicates in actual Datalog programs. **Fact-defined predicates** (or **relations**) are defined by listing all the combinations of values (the tuples) that make the predicate true. These correspond to base relations whose contents are stored in a database system. Figure 26.14 shows the fact-defined predicates EMPLOYEE, MALE, FEMALE, DEPARTMENT, SUPERVISE, PROJECT, and WORKS_ON, which correspond to part of the relational database shown in Figure 3.6. **Rule-defined predicates** (or **views**) are defined by being the head (LHS) of one or more Datalog rules; they correspond to *virtual rela-*

Figure 26.14
Fact predicates for part of the database from Figure 3.6.

```
EMPLOYEE(john).
EMPLOYEE(franklin).
EMPLOYEE(alicia).
EMPLOYEE(jennifer).
EMPLOYEE(ramesh).
EMPLOYEE(joyce).
EMPLOYEE(ahmad).
EMPLOYEE(james).

SALARY(john, 30000).
SALARY(franklin, 40000).
SALARY(alicia, 25000).
SALARY(jennifer, 43000).
SALARY(ramesh, 38000).
SALARY(joyce, 25000).
SALARY(ahmad, 25000).
SALARY(james, 55000).

DEPARTMENT(john, research).
DEPARTMENT(franklin, research).
DEPARTMENT(alicia, administration).
DEPARTMENT(jennifer, administration).
DEPARTMENT(ramesh, research).
DEPARTMENT(joyce, research).
DEPARTMENT(ahmad, administration).
DEPARTMENT(james, headquarters).

SUPERVISE(franklln, john).
SUPERVISE(franklln, ramesh)
SUPERVISE(frankin , joyce).
SUPERVISE(jennifer, alicia).
SUPERVISE(jennifer, ahmad).
SUPERVISE(james, franklin).
SUPERVISE(james, jennifer).

MALE(john).
MALE(franklin).
MALE(ramesh).
MALE(ahmad).
MALE(james).

FEMALE(alicia).
FEMALE(jennifer).
FEMALE(joyce).

PROJECT(productx).
PROJECT(producty).
PROJECT(productz).
PROJECT(computerization).
PROJECT(reorganization).
PROJECT(newbenefits).

WORKS_ON(john, productx, 32).
WORKS_ON(john, producty, 8).
WORKS_ON(ramesh, productz, 40).
WORKS_ON(joyce, productx, 20).
WORKS_ON(joyce, producty, 20).
WORKS_ON(franklin, producty, 10).
WORKS_ON(franklin, productz, 10).
WORKS_ON(franklin, computerization, 10).
WORKS_ON(franklin, reorganization, 10).
WORKS_ON(alicia, newbenefits, 30).
WORKS_ON(alicia, computerization, 10).
WORKS_ON(ahmad, computerization, 35).
WORKS_ON(ahmad, newbenefits, 5).
WORKS_ON(jennifer, newbenefits, 20).
WORKS_ON(jennifer, reorganization, 15).
WORKS_ON(james, reorganization, 10).
```

```
SUPERIOR(X, Y) :– SUPERVISE(X, Y).
SUPERIOR(X, Y) :– SUPERVISE(X, Z), SUPERIOR(Z, Y).

SUBORDINATE(X, Y) :– SUPERIOR(Y, X).

SUPERVISOR(X) :– EMPLOYEE(X), SUPERVISE(X, Y).
OVER_40K_EMP(X) :– EMPLOYEE(X), SALARY(X, Y), Y >= 40000.
UNDER_40K_SUPERVISOR(X) :– SUPERVISOR(X), NOT(OVER_40_K_EMP(X)).
MAIN_PRODUCTX_EMP(X) :– EMPLOYEE(X), WORKS_ON(X, productx, Y), Y >=20.
PRESIDENT(X) :– EMPLOYEE(X), NOT(SUPERVISE(Y, X) ).
```

Figure 26.15
Rule-defined predicates.

tions whose contents can be inferred by the inference engine. Figure 26.15 shows a number of rule-defined predicates.

A program or a rule is said to be **safe** if it generates a *finite* set of facts. The general theoretical problem of determining whether a set of rules is safe is undecidable. However, one can determine the safety of restricted forms of rules. For example, the rules shown in Figure 26.16 are safe. One situation where we get unsafe rules that can generate an infinite number of facts arises when one of the variables in the rule can range over an infinite domain of values, and that variable is not limited to ranging over a finite relation. For example, consider the following rule:

BIG_SALARY(Y) :– Y>60000

Here, we can get an infinite result if Y ranges over all possible integers. But suppose that we change the rule as follows:

BIG_SALARY(Y) :– EMPLOYEE(X), Salary(X, Y), Y>60000

In the second rule, the result is not infinite, since the values that Y can be bound to are now restricted to values that are the salary of some employee in the database—presumably, a finite set of values. We can also rewrite the rule as follows:

BIG_SALARY(Y) :– Y>60000, EMPLOYEE(X), Salary(X, Y)

In this case, the rule is still theoretically safe. However, in Prolog or any other system that uses a top-down, depth-first inference mechanism, the rule creates an infinite loop, since we first search for a value for Y and then check whether it is a salary of an employee. The result is generation of an infinite number of Y values, even though these, after a certain point, cannot lead to a set of true RHS predicates. One definition of Datalog considers both rules to be safe, since it does not depend on a particular inference mechanism. Nonetheless, it is generally advisable to write such a rule in the safest form, with the predicates that restrict possible bindings of variables placed first. As another example of an unsafe rule, consider the following rule:

HAS_SOMETHING(X, Y) :– EMPLOYEE(X)

```
REL_ONE(A, B, C).
REL_TWO(D, E, F).
REL_THREE(G, H, I, J).

SELECT_ONE_A_EQ_C(X, Y, Z) :– REL_ONE(C, Y, Z).
SELECT_ONE_B_LESS_5(X, Y, Z) :– REL_ONE(X, Y, Z), Y< 5.
SELECT_ONE_A_EQ_C_AND_B_LESS_5(X, Y, Z) :– REL_ONE(C, Y, Z), Y<5

SELECT_ONE_A_EQ_C_OR_B_LESS_5(X, Y, Z) :– REL_ONE(C, Y, Z).
SELECT_ONE_A_EQ_C_OR_B_LESS_5(X, Y, Z) :– REL_ONE(X, Y, Z), Y<5.

PROJECT_THREE_ON_G_H(W, X) :– REL_THREE(W, X, Y, Z).

UNION_ONE_TWO(X, Y, Z) :– REL_ONE(X, Y, Z).
UNION_ONE_TWO(X, Y, Z) :– REL_TWO(X, Y, Z).

INTERSECT_ONE_TWO(X, Y, Z) :– REL_ONE(X, Y, Z), REL_TWO(X, Y, Z).

DIFFERENCE_TWO_ONE(X, Y, Z) :– REL_TWO(X, Y, Z) NOT(REL_ONE(X, Y, Z).

CART PROD _ONE_THREE(T, U, V, W, X, Y, Z) :–
    REL_ONE(T, U, V), REL_THREE(W, X, Y, Z).

NATURAL_JOIN_ONE_THREE_C_EQ_G(U, V, W, X, Y, Z) :–
    REL_ONE(U, V, W), REL_THREE(W, X, Y, Z).
```

Figure 26.16
Predicates for illustrating relational operations.

Here, an infinite number of Y values can again be generated, since the variable Y appears only in the head of the rule and hence is not limited to a finite set of values. To define safe rules more formally, we use the concept of a limited variable. A variable X is **limited** in a rule if (1) it appears in a regular (not built-in) predicate in the body of the rule; (2) it appears in a predicate of the form $X=c$ or $c=X$ or ($c_1<=X$ and $X<=c_2$) in the rule body, where c, c_1, and c_2 are constant values; or (3) it appears in a predicate of the form $X=Y$ or $Y=X$ in the rule body, where Y is a limited variable. A rule is said to be **safe** if all its variables are limited.

26.5.7 Use of Relational Operations

It is straightforward to specify many operations of the relational algebra in the form of Datalog rules that define the result of applying these operations on the database relations (fact predicates). This means that relational queries and views can easily be specified in Datalog. The additional power that Datalog provides is in the specification of recursive queries, and views based on recursive queries. In this section, we

show how some of the standard relational operations can be specified as Datalog rules. Our examples will use the base relations (fact-defined predicates) REL_ONE, REL_TWO, and REL_THREE, whose schemas are shown in Figure 26.16. In Datalog, we do not need to specify the attribute names as in Figure 26.16; rather, the arity (degree) of each predicate is the important aspect. In a practical system, the domain (data type) of each attribute is also important for operations such as UNION, INTERSECTION, and JOIN, and we assume that the attribute types are compatible for the various operations, as discussed in Chapter 3.

Figure 26.16 illustrates a number of basic relational operations. Notice that if the Datalog model is based on the relational model and hence assumes that predicates (fact relations and query results) specify sets of tuples, duplicate tuples in the same predicate are automatically eliminated. This may or may not be true, depending on the Datalog inference engine. However, it is definitely *not* the case in Prolog, so any of the rules in Figure 26.16 that involve duplicate elimination are not correct for Prolog. For example, if we want to specify Prolog rules for the UNION operation with duplicate elimination, we must rewrite them as follows:

UNION_ONE_TWO(X, Y, Z) :– REL_ONE(X, Y, Z).

UNION_ONE_TWO(X, Y, Z) :– REL_TWO(X, Y, Z), NOT(REL_ONE(X, Y, Z)).

However, the rules shown in Figure 26.16 should work for Datalog, if duplicates are automatically eliminated. Similarly, the rules for the PROJECT operation shown in Figure 26.16 should work for Datalog in this case, but they are not correct for Prolog, since duplicates would appear in the latter case.

26.5.8 Evaluation of Nonrecursive Datalog Queries

In order to use Datalog as a deductive database system, it is appropriate to define an inference mechanism based on relational database query processing concepts. The inherent strategy involves a bottom-up evaluation, starting with base relations; the order of operations is kept flexible and subject to query optimization. In this section we discuss an **inference mechanism** based on relational operations that can be applied to **nonrecursive** Datalog queries. We use the fact and rule base shown in Figures 26.14 and 26.15 to illustrate our discussion.

If a query involves only fact-defined predicates, the inference becomes one of searching among the facts for the query result. For example, a query such as

DEPARTMENT(X, Research)?

is a selection of all employee names X who work for the Research department. In relational algebra, it is the query:

$$\pi_{\$1}\ (\sigma_{\$2 = \text{“Research”}}\ (\text{DEPARTMENT}))$$

which can be answered by searching through the fact-defined predicate department(X,Y). The query involves relational SELECT and PROJECT operations on a base relation, and it can be handled by the database query processing and optimization techniques discussed in Chapter 19.

When a query involves rule-defined predicates, the inference mechanism must compute the result based on the rule definitions. If a query is nonrecursive and involves a predicate p that appears as the head of a rule $p :- p_1, p_2, ..., p_n$, the strategy is first to compute the relations corresponding to $p_1, p_2, ..., p_n$ and then to compute the relation corresponding to p. It is useful to keep track of the dependency among the predicates of a deductive database in a **predicate dependency graph**. Figure 26.17 shows the graph for the fact and rule predicates shown in Figures 26.14 and 26.15. The dependency graph contains a **node** for each predicate. Whenever a predicate A is specified in the body (RHS) of a rule, and the head (LHS) of that rule is the predicate B, we say that B **depends on** A, and we draw a directed edge from A to B. This indicates that in order to compute the facts for the predicate B (the rule head), we must first compute the facts for all the predicates A in the rule body. If the dependency graph has no cycles, we call the rule set **nonrecursive**. If there is at least one cycle, we call the rule set **recursive**. In Figure 26.17, there is one recursively defined predicate—namely, SUPERIOR—which has a recursive edge pointing back to itself. Additionally, because the predicate subordinate depends on SUPERIOR, it also requires recursion in computing its result.

A query that includes only nonrecursive predicates is called a **nonrecursive query**. In this section we discuss only inference mechanisms for nonrecursive queries. In Figure 26.17, any query that does not involve the predicates SUBORDINATE or SUPERIOR is nonrecursive. In the predicate dependency graph, the nodes corresponding to fact-defined predicates do not have any incoming edges, since all fact-defined predicates have their facts stored in a database relation. The contents of a fact-defined predicate can be computed by directly retrieving the tuples in the corresponding database relation.

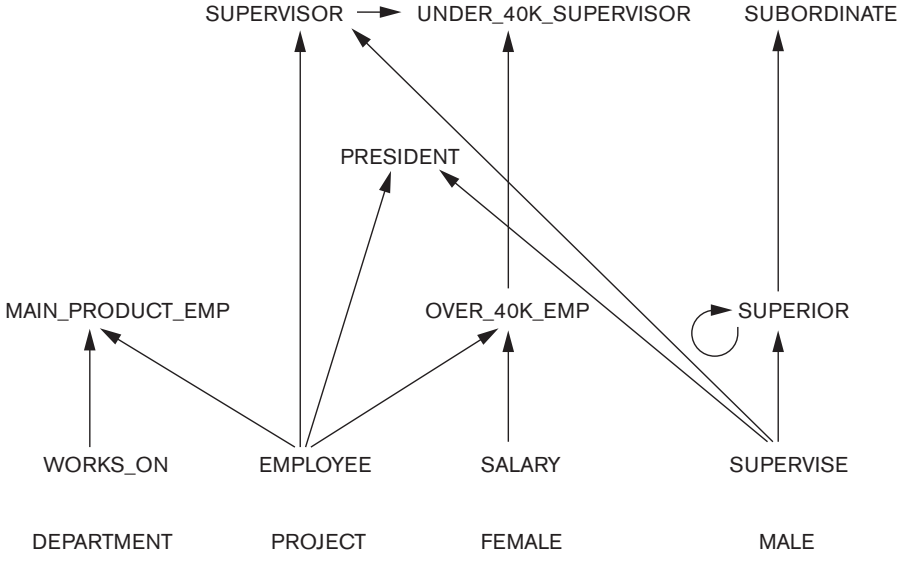


Figure 26.17
Predicate dependency graph for Figures 26.15 and 26.16.

The main function of an inference mechanism is to compute the facts that correspond to query predicates. This can be accomplished by generating a **relational expression** involving relational operators as SELECT, PROJECT, JOIN, UNION, and SET DIFFERENCE (with appropriate provision for dealing with safety issues) that, when executed, provides the query result. The query can then be executed by utilizing the internal query processing and optimization operations of a relational database management system. Whenever the inference mechanism needs to compute the fact set corresponding to a nonrecursive rule-defined predicate p, it first locates all the rules that have p as their head. The idea is to compute the fact set for each such rule and then to apply the UNION operation to the results, since UNION corresponds to a logical OR operation. The dependency graph indicates all predicates q on which each p depends, and since we assume that the predicate is nonrecursive, we can always determine a partial order among such predicates q. Before computing the fact set for p, first we compute the fact sets for all predicates q on which p depends, based on their partial order. For example, if a query involves the predicate UNDER_40K_SUPERVISOR, we must first compute both SUPERVISOR and OVER_40K_EMP. Since the latter two depend only on the fact-defined predicates EMPLOYEE, SALARY, and SUPERVISE, they can be computed directly from the stored database relations.

This concludes our introduction to deductive databases. Additional material may be found at the book's Website, where the complete Chapter 25 from the third edition is available. This includes a discussion on algorithms for recursive query processing. We have included an extensive bibliography of work in deductive databases, recursive query processing, magic sets, combination of relational databases with deductive rules, and GLUE-NAIL! System at the end of this chapter.

26.6 Summary

In this chapter we introduced database concepts for some of the common features that are needed by advanced applications: active databases, temporal databases, spatial databases, multimedia databases, and deductive databases. It is important to note that each of these is a broad topic and warrants a complete textbook.

First we introduced the topic of active databases, which provide additional functionality for specifying active rules. We introduced the Event-Condition-Action (ECA) model for active databases. The rules can be automatically triggered by events that occur—such as a database update—and they can initiate certain actions that have been specified in the rule declaration if certain conditions are true. Many commercial packages have some of the functionality provided by active databases in the form of triggers. We discussed the different options for specifying rules, such as row-level versus statement-level, before versus after, and immediate versus deferred. We gave examples of row-level triggers in the Oracle commercial system, and statement-level rules in the STARBURST experimental system. The syntax for triggers in the SQL-99 standard was also discussed. We briefly discussed some design issues and some possible applications for active databases.

Next we introduced some of the concepts of temporal databases, which permit the database system to store a history of changes and allow users to query both current and past states of the database. We discussed how time is represented and distinguished between the valid time and transaction time dimensions. We discussed how valid time, transaction time, and bitemporal relations can be implemented using tuple versioning in the relational model, with examples to illustrate how updates, inserts, and deletes are implemented. We also showed how complex objects can be used to implement temporal databases using attribute versioning. We looked at some of the querying operations for temporal relational databases and gave a brief introduction to the TSQL2 language.

Then we turned to spatial databases. Spatial databases provide concepts for databases that keep track of objects that have spatial characteristics. We discussed the types of spatial data, types of operators for processing spatial data, types of spatial queries, and spatial indexing techniques, including the popular R-trees. Then we discussed some spatial data mining techniques and applications of spatial data.

We discussed some basic types of multimedia databases and their important characteristics. Multimedia databases provide features that allow users to store and query different types of multimedia information, which includes images (such as pictures and drawings), video clips (such as movies, newsreels, and home videos), audio clips (such as songs, phone messages, and speeches), and documents (such as books and articles). We provided a brief overview of the various types of media sources and how multimedia sources may be indexed. Images are an extremely common type of data among databases today and are likely to occupy a large proportion of stored data in databases. We therefore provided a more detailed treatment of images: their automatic analysis, recognition of objects within images, and their semantic tagging—all of which contribute to developing better systems to retrieve images by content, which still remains a challenging problem. We also commented on the analysis of audio data sources.

We concluded the chapter with an introduction to deductive databases. We gave an overview of Prolog and Datalog notation. We discussed the clausal form of formulas. Datalog rules are restricted to Horn clauses, which contain at most one positive literal. We discussed the proof-theoretic and model-theoretic interpretation of rules. We briefly discussed Datalog rules and their safety and the ways of expressing relational operators using Datalog rules. Finally, we discussed an inference mechanism based on relational operations that can be used to evaluate nonrecursive Datalog queries using relational query optimization techniques. While Datalog has been a popular language with many applications, unfortunately, implementations of deductive database systems such as LDL or VALIDITY have not become widely commercially available.

Review Questions

26.1. What are the differences between row-level and statement-level active rules?

26.2. What are the differences among immediate, deferred, and detached *consideration* of active rule conditions?

26.3. What are the differences among immediate, deferred, and detached *execution* of active rule actions?

26.4. Briefly discuss the consistency and termination problems when designing a set of active rules.

26.5. Discuss some applications of active databases.

26.6. Discuss how time is represented in temporal databases and compare the different time dimensions.

26.7. What are the differences between valid time, transaction time, and bitemporal relations?

26.8. Describe how the insert, delete, and update commands should be implemented on a valid time relation.

26.9. Describe how the insert, delete, and update commands should be implemented on a bitemporal relation.

26.10. Describe how the insert, delete, and update commands should be implemented on a transaction time relation.

26.11. What are the main differences between tuple versioning and attribute versioning?

26.12. How do spatial databases differ from regular databases?

26.13. What are the different types of spatial data?

26.14. Name the main types of spatial operators and different classes of spatial queries.

26.15. What are the properties of R-trees that act as an index for spatial data?

26.16. Describe how a spatial join index between spatial objects can be constructed.

26.17. What are the different types of spatial data mining?

26.18. State the general form of a spatial association rule. Give an example of a spatial association rule.

26.19. What are the different types of multimedia sources?

26.20. How are multimedia sources indexed for content-based retrieval?

26.21. What important features of images are used to compare them?

26.22. What are the different approaches to recognizing objects in images?

26.23. How is semantic tagging of images used?

26.24. What are the difficulties in analyzing audio sources?

26.25. What are deductive databases?

26.26. Write sample rules in Prolog to define that courses with course number above CS5000 are graduate courses and that DBgrads are those graduate students who enroll in CS6400 and CS8803.

26.27. Define clausal form of formulas and Horn clauses.

26.28. What is theorem proving and what is proof-theoretic interpretation of rules?

26.29. What is model-theoretic interpretation and how does it differ from proof-theoretic interpretation?

26.30. What are fact-defined predicates and rule-defined predicates?

26.31. What is a safe rule?

26.32. Give examples of rules that can define relational operations SELECT, PROJECT, JOIN, and SET operations.

26.33. Discuss the inference mechanism based on relational operations that can be applied to evaluate nonrecursive Datalog queries.

Exercises

26.34. Consider the COMPANY database described in Figure 3.6. Using the syntax of Oracle triggers, write active rules to do the following:

a. Whenever an employee's project assignments are changed, check if the total hours per week spent on the employee's projects are less than 30 or greater than 40; if so, notify the employee's direct supervisor.

b. Whenever an employee is deleted, delete the PROJECT tuples and DEPENDENT tuples related to that employee, and if the employee manages a department or supervises employees, set the Mgr_ssn for that department to NULL and set the Super_ssn for those employees to NULL.

26.35. Repeat 26.34 but use the syntax of STARBURST active rules.

26.36. Consider the relational schema shown in Figure 26.18. Write active rules for keeping the Sum_commissions attribute of SALES_PERSON equal to the sum of the Commission attribute in SALES for each sales person. Your rules should also check if the Sum_commissions exceeds 100000; if it does, call a procedure Notify_manager(S_id). Write both statement-level rules in STARBURST notation and row-level rules in Oracle.

SALES

S_id	V_id	Commission

SALES_PERSON

Salesperson_id	Name	Title	Phone	Sum_commissions

Figure 26.18
Database schema for sales and salesperson commissions in Exercise 26.36.

26.37. Consider the UNIVERSITY EER schema in Figure 8.10. Write some rules (in English) that could be implemented via active rules to enforce some common integrity constraints that you think are relevant to this application.

26.38. Discuss which of the updates that created each of the tuples shown in Figure 26.9 were applied retroactively and which were applied proactively.

26.39. Show how the following updates, if applied in sequence, would change the contents of the bitemporal EMP_BT relation in Figure 26.9. For each update, state whether it is a retroactive or proactive update.

a. On 2004-03-10,17:30:00, the salary of Narayan is updated to 40000, effective on 2004-03-01.

b. On 2003-07-30,08:31:00, the salary of Smith was corrected to show that it should have been entered as 31000 (instead of 30000 as shown), effective on 2003-06-01.

c. On 2004-03-18,08:31:00, the database was changed to indicate that Narayan was leaving the company (that is, logically deleted) effective on 2004-03-31.

d. On 2004-04-20,14:07:33, the database was changed to indicate the hiring of a new employee called Johnson, with the tuple <'Johnson', '334455667', 1, NULL > effective on 2004-04-20.

e. On 2004-04-28,12:54:02, the database was changed to indicate that Wong was leaving the company (that is, logically deleted) effective on 2004-06-01.

f. On 2004-05-05,13:07:33, the database was changed to indicate the rehiring of Brown, with the same department and supervisor but with salary 35000 effective on 2004-05-01.

26.40. Show how the updates given in Exercise 26.39, if applied in sequence, would change the contents of the valid time EMP_VT relation in Figure 26.8.

26.41. Add the following facts to the sample database in Figure 26.11:

SUPERVISE(ahmad, bob), SUPERVISE(franklin, gwen).

First modify the supervisory tree in Figure 26.11(b) to reflect this change. Then construct a diagram showing the top-down evaluation of the query SUPERIOR(james, *Y*) using rules 1 and 2 from Figure 26.12.

26.42. Consider the following set of facts for the relation PARENT(X, Y), where Y is the parent of X:

PARENT(a, aa), PARENT(a, ab), PARENT(aa, aaa), PARENT(aa, aab), PARENT(aaa, aaaa), PARENT(aaa, aaab).

Consider the rules

r_1: ANCESTOR(X, Y) :– PARENT(X, Y)
r_2: ANCESTOR(X, Y) :– PARENT(X, Z), ANCESTOR(Z, Y)

which define ancestor Y of X as above.

a. Show how to solve the Datalog query

ANCESTOR(aa, X)?

and show your work at each step.

b. Show the same query by computing only the changes in the ancestor relation and using that in rule 2 each time.

[*This question is derived from Bancilhon and Ramakrishnan (1986).*]

26.43. Consider a deductive database with the following rules:

ANCESTOR(X, Y) :– FATHER(X, Y)
ANCESTOR(X, Y) :– FATHER(X, Z), ANCESTOR(Z, Y)

Notice that FATHER(X, Y) means that Y is the father of X; ANCESTOR(X, Y) means that Y is the ancestor of X.

Consider the following fact base:

FATHER(Harry, Issac), FATHER(Issac, John), FATHER(John, Kurt).

a. Construct a model-theoretic interpretation of the above rules using the given facts.

b. Consider that a database contains the above relations FATHER(X, Y), another relation BROTHER(X, Y), and a third relation BIRTH(X, B), where B is the birth date of person X. State a rule that computes the first cousins of the following variety: their fathers must be brothers.

c. Show a complete Datalog program with fact-based and rule-based literals that computes the following relation: list of pairs of cousins, where the first person is born after 1960 and the second after 1970. You may use *greater than* as a built-in predicate. (*Note*: Sample facts for brother, birth, and person must also be shown.)

26.44. Consider the following rules:

REACHABLE(X, Y) :– FLIGHT(X, Y)
REACHABLE(X, Y) :– FLIGHT(X, Z), REACHABLE(Z, Y)

where REACHABLE(X, Y) means that city Y can be reached from city X, and FLIGHT(X, Y) means that there is a flight to city Y from city X.

a. Construct fact predicates that describe the following:
   i. Los Angeles, New York, Chicago, Atlanta, Frankfurt, Paris, Singapore, Sydney are cities.
   ii. The following flights exist: LA to NY, NY to Atlanta, Atlanta to Frankfurt, Frankfurt to Atlanta, Frankfurt to Singapore, and Singapore to Sydney. (*Note*: No flight in reverse direction can be automatically assumed.)

b. Is the given data cyclic? If so, in what sense?

c. Construct a model-theoretic interpretation (that is, an interpretation similar to the one shown in Figure 26.13) of the above facts and rules.

d. Consider the query

   REACHABLE(Atlanta, Sydney)?

   How will this query be executed? List the series of steps it will go through.

e. Consider the following rule-defined predicates:

   ROUND-TRIP-REACHABLE(X, Y) :–
   REACHABLE(X, Y), REACHABLE(Y, X)
   DURATION(X, Y, Z)

   Draw a predicate dependency graph for the above predicates. (*Note*: DURATION(X, Y, Z) means that you can take a flight from X to Y in Z hours.)

f. Consider the following query: What cities are reachable in 12 hours from Atlanta? Show how to express it in Datalog. Assume built-in predicates like greater-than(X, Y). Can this be converted into a relational algebra statement in a straightforward way? Why or why not?

g. Consider the predicate population(X, Y), where Y is the population of city X. Consider the following query: List all possible bindings of the predicate pair (X, Y), where Y is a city that can be reached in two flights from city X, which has over 1 million people. Show this query in Datalog. Draw a corresponding query tree in relational algebraic terms.

Selected Bibliography

The book by Zaniolo et al. (1997) consists of several parts, each describing an advanced database concept such as active, temporal, and spatial/text/multimedia databases. Widom and Ceri (1996) and Ceri and Fraternali (1997) focus on active database concepts and systems. Snodgrass (1995) describes the TSQL2 language and data model. Khoshafian and Baker (1996), Faloutsos (1996), and Subrahmanian (1998) describe multimedia database concepts. Tansel et al. (1993) is a collection of chapters on temporal databases.

STARBURST rules are described in Widom and Finkelstein (1990). Early work on active databases includes the HiPAC project, discussed in Chakravarthy et al. (1989)

and Chakravarthy (1990). A glossary for temporal databases is given in Jensen et al. (1994). Snodgrass (1987) focuses on TQuel, an early temporal query language.

Temporal normalization is defined in Navathe and Ahmed (1989). Paton (1999) and Paton and Diaz (1999) survey active databases. Chakravarthy et al. (1994) describe SENTINEL and object-based active systems. Lee et al. (1998) discuss time series management.

The book by Shekhar and Chawla (2003) consists of all aspects of spatial databases including spatial data models, spatial storage and indexing, and spatial data mining. Scholl et al. (2001) is another textbook on spatial data management. Albrecht (1996) describes in detail the various GIS analysis operations. Clementini and Di Felice (1993) give a detailed description of the spatial operators. Güting (1994) describes the spatial data structures and querying languages for spatial database systems. Guttman (1984) proposed R-trees for spatial data indexing. Manolopoulos et al. (2005) is a book on the theory and applications of R-trees. Papadias et al. (2003) discuss query processing using R-trees for spatial networks. Ester et al. (2001) provide a comprehensive discussion on the algorithms and applications of spatial data mining. Koperski and Han (1995) discuss association rule discovery from geographic databases. Brinkhoff et al. (1993) provide a comprehensive overview of the usage of R-trees for efficient processing of spatial joins. Rotem (1991) describes spatial join indexes comprehensively. Shekhar and Xiong (2008) is a compilation of various sources that discuss different aspects of spatial database management systems and GIS. The density-based clustering algorithms DBSCAN and DENCLUE are proposed by Ester et al. (1996) and Hinnenberg and Gabriel (2007) respectively.

Multimedia database modeling has a vast amount of literature—it is difficult to point to all important references here. IBM's QBIC (Query By Image Content) system described in Niblack et al. (1998) was one of the first comprehensive approaches for querying images based on content. It is now available as a part of IBM's DB2 database image extender. Zhao and Grosky (2002) discuss content-based image retrieval. Carneiro and Vasconselos (2005) present a database-centric view of semantic image annotation and retrieval. Content-based retrieval of subimages is discussed by Luo and Nascimento (2004). Tuceryan and Jain (1998) discuss various aspects of texture analysis. Object recognition using SIFT is discussed in Lowe (2004). Lazebnik et al. (2004) describe the use of local affine regions to model 3D objects (RIFT). Among other object recognition approaches, G-RIF is described in Kim et al. (2006), Bay et al. (2006) discuss SURF, Ke and Sukthankar (2004) present PCA-SIFT, and Mikolajczyk and Schmid (2005) describe GLOH. Fan et al. (2004) present a technique for automatic image annotation by using concept-sensitive objects. Fotouhi et al. (2007) was the first international workshop on many faces of multimedia semantics, which is continuing annually. Thuraisingham (2001) classifies audio data into different categories, and by treating each of these categories differently, elaborates on the use of metadata for audio. Prabhakaran (1996) has also discussed how speech processing techniques can add valuable metadata information to the audio piece.

The early developments of the logic and database approach are surveyed by Gallaire et al. (1984). Reiter (1984) provides a reconstruction of relational database theory,

while Levesque (1984) provides a discussion of incomplete knowledge in light of logic. Gallaire and Minker (1978) provide an early book on this topic. A detailed treatment of logic and databases appears in Ullman (1989, Volume 2), and there is a related chapter in Volume 1 (1988). Ceri, Gottlob, and Tanca (1990) present a comprehensive yet concise treatment of logic and databases. Das (1992) is a comprehensive book on deductive databases and logic programming. The early history of Datalog is covered in Maier and Warren (1988). Clocksin and Mellish (2003) is an excellent reference on Prolog language.

Aho and Ullman (1979) provide an early algorithm for dealing with recursive queries, using the least fixed-point operator. Bancilhon and Ramakrishnan (1986) give an excellent and detailed description of the approaches to recursive query processing, with detailed examples of the naive and seminaive approaches. Excellent survey articles on deductive databases and recursive query processing include Warren (1992) and Ramakrishnan and Ullman (1995). A complete description of the seminaive approach based on relational algebra is given in Bancilhon (1985). Other approaches to recursive query processing include the recursive query/subquery strategy of Vieille (1986), which is a top-down interpreted strategy, and the Henschen-Naqvi (1984) top-down compiled iterative strategy. Balbin and Ramamohanrao (1987) discuss an extension of the seminaive differential approach for multiple predicates.

The original paper on magic sets is by Bancilhon et al. (1986). Beeri and Ramakrishnan (1987) extend it. Mumick et al. (1990a) show the applicability of magic sets to nonrecursive nested SQL queries. Other approaches to optimizing rules without rewriting them appear in Vieille (1986, 1987). Kifer and Lozinskii (1986) propose a different technique. Bry (1990) discusses how the top-down and bottom-up approaches can be reconciled. Whang and Navathe (1992) describe an extended disjunctive normal form technique to deal with recursion in relational algebra expressions for providing an expert system interface over a relational DBMS.

Chang (1981) describes an early system for combining deductive rules with relational databases. The LDL system prototype is described in Chimenti et al. (1990). Krishnamurthy and Naqvi (1989) introduce the *choice* notion in LDL. Zaniolo (1988) discusses the language issues for the LDL system. A language overview of CORAL is provided in Ramakrishnan et al. (1992), and the implementation is described in Ramakrishnan et al. (1993). An extension to support object-oriented features, called CORAL++, is described in Srivastava et al. (1993). Ullman (1985) provides the basis for the NAIL! system, which is described in Morris et al. (1987). Phipps et al. (1991) describe the GLUE-NAIL! deductive database system.

Zaniolo (1990) reviews the theoretical background and the practical importance of deductive databases. Nicolas (1997) gives an excellent history of the developments leading up to Deductive Object-Oriented Database (DOOD) systems. Falcone et al. (1997) survey the DOOD landscape. References on the VALIDITY system include Friesen et al. (1995), Vieille (1998), and Dietrich et al. (1999).

chapter 28

Data Mining Concepts

Over the last three decades, many organizations have generated a large amount of machine-readable data in the form of files and databases. To process this data, we have the database technology available that supports query languages like SQL. The problem with SQL is that it is a structured language that assumes the user is aware of the database schema. SQL supports operations of relational algebra that allow a user to select rows and columns of data from tables or join-related information from tables based on common fields. In the next chapter, we will see that *data warehousing technology* affords several types of functionality: that of consolidation, aggregation, and summarization of data. Data warehouses let us view the same information along multiple dimensions. In this chapter, we will focus our attention on another very popular area of interest known as data mining. As the term connotes, **data mining** refers to the mining or discovery of new information in terms of patterns or rules from vast amounts of data. To be practically useful, data mining must be carried out efficiently on large files and databases. Although some data mining features are being provided in RDBMSs, data mining is *not* well-integrated with database management systems.

We will briefly review the state of the art of this rather extensive field of data mining, which uses techniques from such areas as machine learning, statistics, neural networks, and genetic algorithms. We will highlight the nature of the information that is discovered, the types of problems faced when trying to mine databases, and the types of applications of data mining. We will also survey the state of the art of a large number of commercial tools available (see Section 28.7) and describe a number of research advances that are needed to make this area viable.

28.1 Overview of Data Mining Technology

In reports such as the very popular Gartner Report,[1] data mining has been hailed as one of the top technologies for the near future. In this section we relate data mining to the broader area called *knowledge discovery* and contrast the two by means of an illustrative example.

28.1.1 Data Mining versus Data Warehousing

The goal of a data warehouse (see Chapter 29) is to support decision making with data. Data mining can be used in conjunction with a data warehouse to help with certain types of decisions. Data mining can be applied to operational databases with individual transactions. To make data mining more efficient, the data warehouse should have an aggregated or summarized collection of data. Data mining helps in extracting meaningful new patterns that cannot necessarily be found by merely querying or processing data or metadata in the data warehouse. Therefore, data mining applications should be strongly considered early, during the design of a data warehouse. Also, data mining tools should be designed to facilitate their use in conjunction with data warehouses. In fact, for very large databases running into terabytes and even petabytes of data, successful use of data mining applications will depend first on the construction of a data warehouse.

28.1.2 Data Mining as a Part of the Knowledge Discovery Process

Knowledge Discovery in Databases, frequently abbreviated as **KDD**, typically encompasses more than data mining. The knowledge discovery process comprises six phases:[2] data selection, data cleansing, enrichment, data transformation or encoding, data mining, and the reporting and display of the discovered information.

As an example, consider a transaction database maintained by a specialty consumer goods retailer. Suppose the client data includes a customer name, ZIP Code, phone number, date of purchase, item code, price, quantity, and total amount. A variety of new knowledge can be discovered by KDD processing on this client database. During *data selection,* data about specific items or categories of items, or from stores in a specific region or area of the country, may be selected. The *data cleansing* process then may correct invalid ZIP Codes or eliminate records with incorrect phone prefixes. *Enrichment* typically enhances the data with additional sources of information. For example, given the client names and phone numbers, the store may purchase other data about age, income, and credit rating and append them to each record. *Data transformation* and encoding may be done to reduce the amount

[1]The Gartner Report is one example of the many technology survey publications that corporate managers rely on to make their technology selection discussions.

[2]This discussion is largely based on Adriaans and Zantinge (1996).

of data. For instance, item codes may be grouped in terms of product categories into audio, video, supplies, electronic gadgets, camera, accessories, and so on. ZIP Codes may be aggregated into geographic regions, incomes may be divided into ranges, and so on. In Figure 29.1, we will show a step called *cleaning* as a precursor to the data warehouse creation. If data mining is based on an existing warehouse for this retail store chain, we would expect that the cleaning has already been applied. It is only after such preprocessing that *data mining* techniques are used to mine different rules and patterns.

The result of mining may be to discover the following type of *new* information:

- **Association rules**—for example, whenever a customer buys video equipment, he or she also buys another electronic gadget.
- **Sequential patterns**—for example, suppose a customer buys a camera, and within three months he or she buys photographic supplies, then within six months he is likely to buy an accessory item. This defines a sequential pattern of transactions. A customer who buys more than twice in lean periods may be likely to buy at least once during the Christmas period.
- **Classification trees**—for example, customers may be classified by frequency of visits, types of financing used, amount of purchase, or affinity for types of items; some revealing statistics may be generated for such classes.

We can see that many possibilities exist for discovering new knowledge about buying patterns, relating factors such as age, income group, place of residence, to what and how much the customers purchase. This information can then be utilized to plan additional store locations based on demographics, run store promotions, combine items in advertisements, or plan seasonal marketing strategies. As this retail store example shows, data mining must be preceded by significant data preparation before it can yield useful information that can directly influence business decisions.

The results of data mining may be reported in a variety of formats, such as listings, graphic outputs, summary tables, or visualizations.

28.1.3 Goals of Data Mining and Knowledge Discovery

Data mining is typically carried out with some end goals or applications. Broadly speaking, these goals fall into the following classes: prediction, identification, classification, and optimization.

- **Prediction.** Data mining can show how certain attributes within the data will behave in the future. Examples of predictive data mining include the analysis of buying transactions to predict what consumers will buy under certain discounts, how much sales volume a store will generate in a given period, and whether deleting a product line will yield more profits. In such applications, business logic is used coupled with data mining. In a scientific context, certain seismic wave patterns may predict an earthquake with high probability.

- **Identification.** Data patterns can be used to identify the existence of an item, an event, or an activity. For example, intruders trying to break a system may be identified by the programs executed, files accessed, and CPU time per session. In biological applications, existence of a gene may be identified by certain sequences of nucleotide symbols in the DNA sequence. The area known as *authentication* is a form of identification. It ascertains whether a user is indeed a specific user or one from an authorized class, and involves a comparison of parameters or images or signals against a database.
- **Classification.** Data mining can partition the data so that different classes or categories can be identified based on combinations of parameters. For example, customers in a supermarket can be categorized into discount-seeking shoppers, shoppers in a rush, loyal regular shoppers, shoppers attached to name brands, and infrequent shoppers. This classification may be used in different analyses of customer buying transactions as a post-mining activity. Sometimes classification based on common domain knowledge is used as an input to decompose the mining problem and make it simpler. For instance, health foods, party foods, or school lunch foods are distinct categories in the supermarket business. It makes sense to analyze relationships within and across categories as separate problems. Such categorization may be used to encode the data appropriately before subjecting it to further data mining.
- **Optimization.** One eventual goal of data mining may be to optimize the use of limited resources such as time, space, money, or materials and to maximize output variables such as sales or profits under a given set of constraints. As such, this goal of data mining resembles the objective function used in operations research problems that deals with optimization under constraints.

The term data mining is popularly used in a very broad sense. In some situations it includes statistical analysis and constrained optimization as well as machine learning. There is no sharp line separating data mining from these disciplines. It is beyond our scope, therefore, to discuss in detail the entire range of applications that make up this vast body of work. For a detailed understanding of the topic, readers are referred to specialized books devoted to data mining.

28.1.4 Types of Knowledge Discovered during Data Mining

The term *knowledge* is broadly interpreted as involving some degree of intelligence. There is a progression from raw data to information to knowledge as we go through additional processing. Knowledge is often classified as inductive versus deductive. **Deductive knowledge** deduces new information based on applying *prespecified* logical rules of deduction on the given data. Data mining addresses **inductive knowledge**, which discovers new rules and patterns from the supplied data. Knowledge can be represented in many forms: In an unstructured sense, it can be represented by rules or propositional logic. In a structured form, it may be represented in deci-

sion trees, semantic networks, neural networks, or hierarchies of classes or frames. It is common to describe the knowledge discovered during data mining as follows:

- **Association rules.** These rules correlate the presence of a set of items with another range of values for another set of variables. Examples: (1) When a female retail shopper buys a handbag, she is likely to buy shoes. (2) An X-ray image containing characteristics a and b is likely to also exhibit characteristic c.
- **Classification hierarchies.** The goal is to work from an existing set of events or transactions to create a hierarchy of classes. Examples: (1) A population may be divided into five ranges of credit worthiness based on a history of previous credit transactions. (2) A model may be developed for the factors that determine the desirability of a store location on a 1–10 scale. (3) Mutual funds may be classified based on performance data using characteristics such as growth, income, and stability.
- **Sequential patterns.** A sequence of actions or events is sought. Example: If a patient underwent cardiac bypass surgery for blocked arteries and an aneurysm and later developed high blood urea within a year of surgery, he or she is likely to suffer from kidney failure within the next 18 months. Detection of sequential patterns is equivalent to detecting associations among events with certain temporal relationships.
- **Patterns within time series.** Similarities can be detected within positions of a **time series** of data, which is a sequence of data taken at regular intervals, such as daily sales or daily closing stock prices. Examples: (1) Stocks of a utility company, ABC Power, and a financial company, XYZ Securities, showed the same pattern during 2009 in terms of closing stock prices. (2) Two products show the same selling pattern in summer but a different one in winter. (3) A pattern in solar magnetic wind may be used to predict changes in Earth's atmospheric conditions.
- **Clustering.** A given population of events or items can be partitioned (segmented) into sets of "similar" elements. Examples: (1) An entire population of treatment data on a disease may be divided into groups based on the similarity of side effects produced. (2) The adult population in the United States may be categorized into five groups from *most likely to buy* to *least likely to buy* a new product. (3) The Web accesses made by a collection of users against a set of documents (say, in a digital library) may be analyzed in terms of the keywords of documents to reveal clusters or categories of users.

For most applications, the desired knowledge is a combination of the above types. We expand on each of the above knowledge types in the following sections.

28.2 Association Rules

28.2.1 Market-Basket Model, Support, and Confidence

One of the major technologies in data mining involves the discovery of association rules. The database is regarded as a collection of transactions, each involving a set of

items. A common example is that of **market-basket data**. Here the market basket corresponds to the sets of items a consumer buys in a supermarket during one visit. Consider four such transactions in a random sample shown in Figure 28.1.

An **association rule** is of the form $X => Y$, where $X = \{x_1, x_2, ..., x_n\}$, and $Y = \{y_1, y_2, ..., y_m\}$ are sets of items, with x_i and y_j being distinct items for all i and all j. This association states that if a customer buys X, he or she is also likely to buy Y. In general, any association rule has the form LHS (left-hand side) => RHS (right-hand side), where LHS and RHS are sets of items. The set LHS $\cup$ RHS is called an **itemset**, the set of items purchased by customers. For an association rule to be of interest to a data miner, the rule should satisfy some interest measure. Two common interest measures are support and confidence.

The **support** for a rule LHS => RHS is with respect to the itemset; it refers to how frequently a specific itemset occurs in the database. That is, the support is the percentage of transactions that contain all of the items in the itemset LHS $\cup$ RHS. If the support is low, it implies that there is no overwhelming evidence that items in LHS $\cup$ RHS occur together because the itemset occurs in only a small fraction of transactions. Another term for support is *prevalence* of the rule.

The **confidence** is with regard to the implication shown in the rule. The confidence of the rule LHS => RHS is computed as the support(LHS $\cup$ RHS)/support(LHS). We can think of it as the probability that the items in RHS will be purchased given that the items in LHS are purchased by a customer. Another term for confidence is *strength* of the rule.

As an example of support and confidence, consider the following two rules: milk => juice and bread => juice. Looking at our four sample transactions in Figure 28.1, we see that the support of {milk, juice} is 50 percent and the support of {bread, juice} is only 25 percent. The confidence of milk => juice is 66.7 percent (meaning that, of three transactions in which milk occurs, two contain juice) and the confidence of bread => juice is 50 percent (meaning that one of two transactions containing bread also contains juice).

As we can see, support and confidence do not necessarily go hand in hand. The goal of mining association rules, then, is to generate all possible rules that exceed some minimum user-specified support and confidence thresholds. The problem is thus decomposed into two subproblems:

1. Generate all itemsets that have a support that exceeds the threshold. These sets of items are called **large** (or **frequent**) **itemsets**. Note that large here means large support.

Figure 28.1
Sample transactions in market-basket model.

Transaction_id	Time	Items_bought
101	6:35	milk, bread, cookies, juice
792	7:38	milk, juice
1130	8:05	milk, eggs
1735	8:40	bread, cookies, coffee

2. For each large itemset, all the rules that have a minimum confidence are generated as follows: For a large itemset X and $Y \subset X$, let $Z = X - Y$; then if support(X)/support(Z) > minimum confidence, the rule $Z \Rightarrow Y$ (that is, $X - Y \Rightarrow Y$) is a valid rule.

Generating rules by using all large itemsets and their supports is relatively straightforward. However, discovering all large itemsets together with the value for their support is a major problem if the cardinality of the set of items is very high. A typical supermarket has thousands of items. The number of distinct itemsets is 2^m, where m is the number of items, and counting support for all possible itemsets becomes very computation intensive. To reduce the combinatorial search space, algorithms for finding association rules utilize the following properties:

- A subset of a large itemset must also be large (that is, each subset of a large itemset exceeds the minimum required support).
- Conversely, a superset of a small itemset is also small (implying that it does not have enough support).

The first property is referred to as **downward closure**. The second property, called the **antimonotonicity** property, helps to reduce the search space of possible solutions. That is, once an itemset is found to be small (not a large itemset), then any extension to that itemset, formed by adding one or more items to the set, will also yield a small itemset.

28.2.2 Apriori Algorithm

The first algorithm to use the downward closure and antimontonicity properties was the **Apriori algorithm**, shown as Algorithm 28.1.

We illustrate Algorithm 28.1 using the transaction data in Figure 28.1 using a minimum support of 0.5. The candidate 1-itemsets are {milk, bread, juice, cookies, eggs, coffee} and their respective supports are 0.75, 0.5, 0.5, 0.5, 0.25, and 0.25. The first four items qualify for L_1 since each support is greater than or equal to 0.5. In the first iteration of the repeat-loop, we extend the frequent 1-itemsets to create the candidate frequent 2-itemsets, C_2. C_2 contains {milk, bread}, {milk, juice}, {bread, juice}, {milk, cookies}, {bread, cookies}, and {juice, cookies}. Notice, for example, that {milk, eggs} does not appear in C_2 since {eggs} is small (by the antimonotonicity property) and does not appear in L_1. The supports for the six sets contained in C_2 are 0.25, 0.5, 0.25, 0.25, 0.5, and 0.25 and are computed by scanning the set of transactions. Only the second 2-itemset {milk, juice} and the fifth 2-itemset {bread, cookies} have support greater than or equal to 0.5. These two 2-itemsets form the frequent 2-itemsets, L_2.

Algorithm 28.1. Apriori Algorithm for Finding Frequent (Large) Itemsets

Input: Database of m transactions, D, and a minimum support, *mins*, represented as a fraction of m.

Output: Frequent itemsets, $L_1, L_2, \ldots, L_k$

Begin /* steps or statements are numbered for better readability */

1. Compute support(i_j) = count(i_j)/m for each individual item, $i_1, i_2, \ldots, i_n$ by scanning the database once and counting the number of transactions that item i_j appears in (that is, count(i_j));
2. The candidate frequent 1-itemset, C_1, will be the set of items $i_1, i_2, \ldots, i_n$;
3. The subset of items containing i_j from C_1 where support(i_j) >= mins becomes the frequent

   1-itemset, L_1;
4. $k = 1$;

   termination = false;

   repeat

1. $L_{k+1} =$;
2. Create the candidate frequent (k+1)-itemset, C_{k+1}, by combining members of L_k that have k–1 items in common (this forms candidate frequent (k+1)-itemsets by selectively extending frequent k-itemsets by one item);
3. In addition, only consider as elements of C_{k+1} those k+1 items such that every subset of size k appears in L_k;
4. Scan the database once and compute the support for each member of C_{k+1}; if the support for a member of C_{k+1} >= mins then add that member to L_{k+1};
5. If L_{k+1} is empty then termination = true

   else $k = k + 1$;

   until termination;

End;

In the next iteration of the repeat-loop, we construct candidate frequent 3-itemsets by adding additional items to sets in L_2. However, for no extension of itemsets in L_2 will all 2-item subsets be contained in L_2. For example, consider {milk, juice, bread}; the 2-itemset {milk, bread} is not in L_2, hence {milk, juice, bread} cannot be a frequent 3-itemset by the downward closure property. At this point the algorithm terminates with L_1 equal to {{milk}, {bread}, {juice}, {cookies}} and L_2 equal to {{milk, juice}, {bread, cookies}}.

Several other algorithms have been proposed to mine association rules. They vary mainly in terms of how the candidate itemsets are generated, and how the supports for the candidate itemsets are counted. Some algorithms use such data structures as bitmaps and hashtrees to keep information about itemsets. Several algorithms have been proposed that use multiple scans of the database because the potential number of itemsets, 2^m, can be too large to set up counters during a single scan. We will examine three improved algorithms (compared to the Apriori algorithm) for association rule mining: the Sampling algorithm, the Frequent-Pattern Tree algorithm, and the Partition algorithm.

28.2.3 Sampling Algorithm

The main idea for the **Sampling algorithm** is to select a small sample, one that fits in main memory, of the database of transactions and to determine the frequent itemsets from that sample. If those frequent itemsets form a superset of the frequent itemsets for the entire database, then we can determine the real frequent itemsets by scanning the remainder of the database in order to compute the exact support values for the superset itemsets. A superset of the frequent itemsets can usually be found from the sample by using, for example, the Apriori algorithm, with a lowered minimum support.

In some rare cases, some frequent itemsets may be missed and a second scan of the database is needed. To decide whether any frequent itemsets have been missed, the concept of the *negative border* is used. The negative border with respect to a frequent itemset, S, and set of items, I, is the minimal itemsets contained in PowerSet(I) and not in S. The basic idea is that the negative border of a set of frequent itemsets contains the closest itemsets that could also be frequent. Consider the case where a set X is not contained in the frequent itemsets. If all subsets of X are contained in the set of frequent itemsets, then X would be in the negative border.

We illustrate this with the following example. Consider the set of items I = {A, B, C, D, E} and let the combined frequent itemsets of size 1 to 3 be S = {{A}, {B}, {C}, {D}, {AB}, {AC}, {BC}, {AD}, {CD}, {ABC}}. The negative border is {{E}, {BD}, {ACD}}. The set {E} is the only 1-itemset not contained in S, {BD} is the only 2-itemset not in S but whose 1-itemset subsets are, and {ACD} is the only 3-itemset whose 2-itemset subsets are all in S. The negative border is important since it is necessary to determine the support for those itemsets in the negative border to ensure that no large itemsets are missed from analyzing the sample data.

Support for the negative border is determined when the remainder of the database is scanned. If we find that an itemset, X, in the negative border belongs in the set of all frequent itemsets, then there is a potential for a superset of X to also be frequent. If this happens, then a second pass over the database is needed to make sure that all frequent itemsets are found.

28.2.4 Frequent-Pattern (FP) Tree and FP-Growth Algorithm

The **Frequent-Pattern Tree** (**FP-tree**) is motivated by the fact that Apriori-based algorithms may generate and test a very large number of candidate itemsets. For example, with 1000 frequent 1-itemsets, the Apriori algorithm would have to generate

$$\binom{1000}{2}$$

or 499,500 candidate 2-itemsets. The **FP-Growth algorithm** is one approach that eliminates the generation of a large number of candidate itemsets.

The algorithm first produces a compressed version of the database in terms of an FP-tree (frequent-pattern tree). The FP-tree stores relevant itemset information and allows for the efficient discovery of frequent itemsets. The actual mining process adopts a divide-and-conquer strategy where the mining process is decomposed into a set of smaller tasks that each operates on a conditional FP-tree, a subset (projection) of the original tree. To start with, we examine how the FP-tree is constructed. The database is first scanned and the frequent 1-itemsets along with their support are computed. With this algorithm, the support is the *count* of transactions containing the item rather than the fraction of transactions containing the item. The frequent 1-itemsets are then sorted in nonincreasing order of their support. Next, the root of the FP-tree is created with a NULL label. The database is scanned a second time and for each transaction T in the database, the frequent 1-itemsets in T are placed in order as was done with the frequent 1-itemsets. We can designate this sorted list for T as consisting of a first item, the head, and the remaining items, the tail. The itemset information (head, tail) is inserted into the FP-tree recursively, starting at the root node, as follows:

1. If the current node, N, of the FP-tree has a child with an item name = head, then increment the count associated with node N by 1, else create a new node, N, with a count of 1, link N to its parent and link N with the item header table (used for efficient tree traversal).
2. If the tail is nonempty, then repeat step (1) using as the sorted list only the tail, that is, the old head is removed and the new head is the first item from the tail and the remaining items become the new tail.

The item header table, created during the process of building the FP-tree, contains three fields per entry for each frequent item: item identifier, support count, and node link. The item identifier and support count are self-explanatory. The node link is a pointer to an occurrence of that item in the FP-tree. Since multiple occurrences of a single item may appear in the FP-tree, these items are linked together as a list where the start of the list is pointed to by the node link in the item header table. We illustrate the building of the FP-tree using the transaction data in Figure 28.1. Let us use a minimum support of 2. One pass over the four transactions yields the following frequent 1-itemsets with associated support: {{(milk, 3)}, {(bread, 2)}, {(cookies, 2)}, {(juice, 2)}}. The database is scanned a second time and each transaction will be processed again.

For the first transaction, we create the sorted list, T = {milk, bread, cookies, juice}. The items in T are the frequent 1-itemsets from the first transaction. The items are ordered based on the nonincreasing ordering of the count of the 1-itemsets found in pass 1 (that is, milk first, bread second, and so on). We create a NULL root node for the FP-tree and insert *milk* as a child of the root, *bread* as a child of *milk*, *cookies* as a child of *bread*, and *juice* as a child of *cookies*. We adjust the entries for the frequent items in the item header table.

For the second transaction, we have the sorted list {milk, juice}. Starting at the root, we see that a child node with label *milk* exists, so we move to that node and update

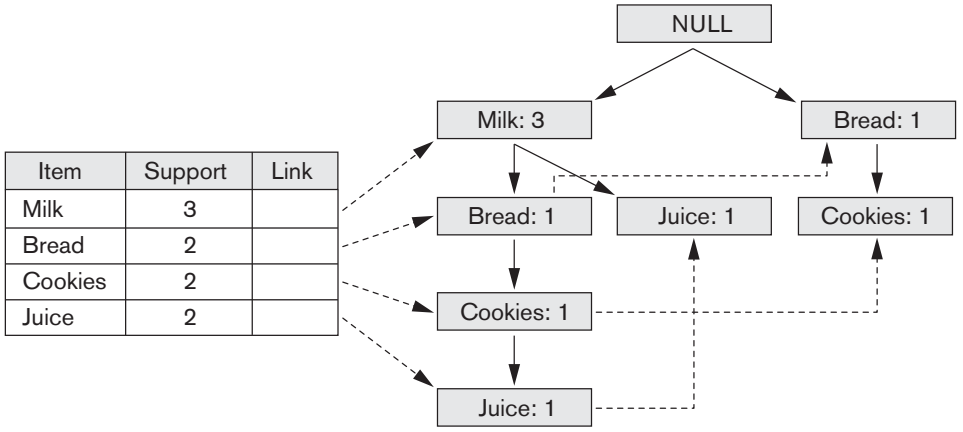


Figure 28.2
FP-tree and item header table.

its count (to account for the second transaction that contains milk). We see that there is no child of the current node with label *juice*, so we create a new node with label *juice*. The item header table is adjusted.

The third transaction only has 1-frequent item, {milk}. Again, starting at the root, we see that the node with label *milk* exists, so we move to that node, increment its count, and adjust the item header table. The final transaction contains frequent items, {bread, cookies}. At the root node, we see that a child with label *bread* does not exist. Thus, we create a new child of the root, initialize its counter, and then insert *cookies* as a child of this node and initialize its count. After the item header table is updated, we end up with the FP-tree and item header table as shown in Figure 28.2. If we examine this FP-tree, we see that it indeed represents the original transactions in a compressed format (that is, only showing the items from each transaction that are large 1-itemsets).

Algorithm 28.2 is used for mining the FP-tree for frequent patterns. With the FP-tree, it is possible to find all frequent patterns that contain a given frequent item by starting from the item header table for that item and traversing the node links in the FP-tree. The algorithm starts with a frequent 1-itemset (suffix pattern) and constructs its conditional pattern base and then its conditional FP-tree. The conditional pattern base is made up of a set of prefix paths, that is, where the frequent item is a suffix. For example, if we consider the item juice, we see from Figure 28.2 that there are two paths in the FP-tree that end with juice: (milk, bread, cookies, juice) and (milk, juice). The two associated prefix paths are (milk, bread, cookies) and (milk). The conditional FP-tree is constructed from the patterns in the conditional pattern base. The mining is recursively performed on this FP-tree. The frequent patterns are formed by concatenating the suffix pattern with the frequent patterns produced from a conditional FP-tree.

Algorithm 28.2. FP-Growth Algorithm for Finding Frequent Itemsets

Input: FP-tree and a minimum support, mins

Output: frequent patterns (itemsets)

procedure FP-growth (tree, alpha);

Begin
    if tree contains a single path *P* then
        for each combination, beta, of the nodes in the path
            generate pattern (beta ∪ alpha)
            with support = minimum support of nodes in beta
    else
        for each item, *i*, in the header of the tree do
            begin
                generate pattern beta = (*i* ∪ alpha) with support = *i*.support;
                construct beta's conditional pattern base;
                construct beta's conditional FP-tree, beta_tree;
                if beta_tree is not empty then
                    FP-growth(beta_tree, beta);
            end;
End;

We illustrate the algorithm using the data in Figure 28.1 and the tree in Figure 28.2. The procedure FP-growth is called with the two parameters: the original FP-tree and NULL for the variable alpha. Since the original FP-tree has more than a single path, we execute the else part of the first if statement. We start with the frequent item, juice. We will examine the frequent items in order of lowest support (that is, from the last entry in the table to the first). The variable beta is set to juice with support equal to 2.

Following the node link in the item header table, we construct the conditional pattern base consisting of two paths (with juice as suffix). These are (milk, bread, cookies: 1) and (milk: 1). The conditional FP-tree consists of only a single node, milk: 2. This is due to a support of only 1 for node bread and cookies, which is below the minimal support of 2. The algorithm is called recursively with an FP-tree of only a single node (that is, milk: 2) and a beta value of juice. Since this FP-tree only has one path, all combinations of beta and nodes in the path are generated—that is, {milk, juice}—with support of 2.

Next, the frequent item, cookies, is used. The variable beta is set to cookies with support = 2. Following the node link in the item header table, we construct the conditional pattern base consisting of two paths. These are (milk, bread: 1) and (bread: 1). The conditional FP-tree is only a single node, bread: 2. The algorithm is called recursively with an FP-tree of only a single node (that is, bread: 2) and a beta value of cookies. Since this FP-tree only has one path, all combinations of beta and nodes in the path are generated, that is, {bread, cookies} with support of 2. The frequent item, bread, is considered next. The variable beta is set to bread with support = 2. Following the node link in the item header table, we construct the conditional

pattern base consisting of one path, which is (milk: 1). The conditional FP-tree is empty since the count is less than the minimum support. Since the conditional FP-tree is empty, no frequent patterns will be generated.

The last frequent item to consider is milk. This is the top item in the item header table and as such has an empty conditional pattern base and empty conditional FP-tree. As a result, no frequent patterns are added. The result of executing the algorithm is the following frequent patterns (or itemsets) with their support: {{milk: 3}, {bread: 2}, {cookies: 2}, {juice: 2}, {milk, juice: 2}, {bread, cookies: 2}}.

28.2.5 Partition Algorithm

Another algorithm, called the **Partition algorithm**,[3] is summarized below. If we are given a database with a small number of potential large itemsets, say, a few thousand, then the support for all of them can be tested in one scan by using a partitioning technique. Partitioning divides the database into nonoverlapping subsets; these are individually considered as separate databases and all large itemsets for that partition, called *local frequent itemsets*, are generated in one pass. The Apriori algorithm can then be used efficiently on each partition if it fits entirely in main memory. Partitions are chosen in such a way that each partition can be accommodated in main memory. As such, a partition is read only once in each pass. The only caveat with the partition method is that the minimum support used for each partition has a slightly different meaning from the original value. The minimum support is based on the size of the partition rather than the size of the database for determining local frequent (large) itemsets. The actual support threshold value is the same as given earlier, but the support is computed only for a partition.

At the end of pass one, we take the union of all frequent itemsets from each partition. This forms the global candidate frequent itemsets for the entire database. When these lists are merged, they may contain some false positives. That is, some of the itemsets that are frequent (large) in one partition may not qualify in several other partitions and hence may not exceed the minimum support when the original database is considered. Note that there are no false negatives; no large itemsets will be missed. The global candidate large itemsets identified in pass one are verified in pass two; that is, their actual support is measured for the *entire* database. At the end of phase two, all global large itemsets are identified. The Partition algorithm lends itself naturally to a parallel or distributed implementation for better efficiency. Further improvements to this algorithm have been suggested.[4]

28.2.6 Other Types of Association Rules

Association Rules among Hierarchies. There are certain types of associations that are particularly interesting for a special reason. These associations occur among

[3]See Savasere et al. (1995) for details of the algorithm, the data structures used to implement it, and its performance comparisons.

[4]See Cheung et al. (1996) and Lin and Dunham (1998).

hierarchies of items. Typically, it is possible to divide items among disjoint hierarchies based on the nature of the domain. For example, foods in a supermarket, items in a department store, or articles in a sports shop can be categorized into classes and subclasses that give rise to hierarchies. Consider Figure 28.3, which shows the taxonomy of items in a supermarket. The figure shows two hierarchies—beverages and desserts, respectively. The entire groups may not produce associations of the form beverages => desserts, or desserts => beverages. However, associations of the type Healthy-brand frozen yogurt => bottled water, or Rich cream-brand ice cream => wine cooler may produce enough confidence and support to be valid association rules of interest.

Therefore, if the application area has a natural classification of the itemsets into hierarchies, discovering associations *within* the hierarchies is of no particular interest. The ones of specific interest are associations *across* hierarchies. They may occur among item groupings at different levels.

Multidimensional Associations. Discovering association rules involves searching for patterns in a file. In Figure 28.1, we have an example of a file of customer transactions with three dimensions: Transaction_id, Time, and Items_bought. However, our data mining tasks and algorithms introduced up to this point only involve one dimension: Items_bought. The following rule is an example of including the label of the single dimension: Items_bought(milk) => Items_bought(juice). It may be of interest to find association rules that involve multiple dimensions, for

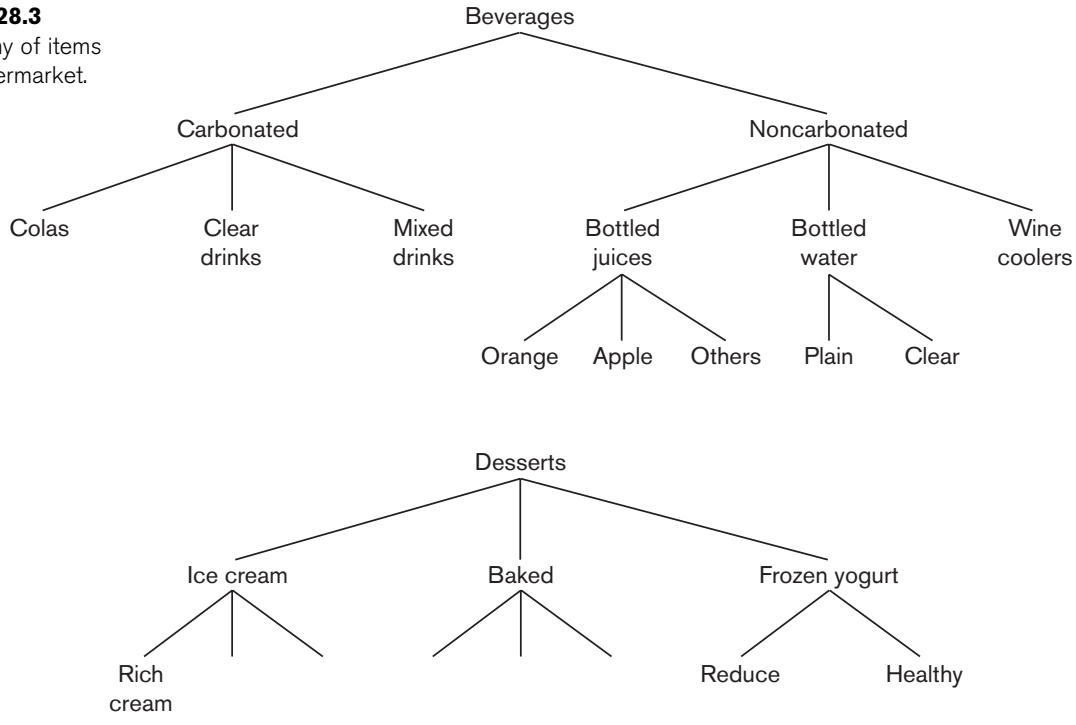


Figure 28.3
Taxonomy of items in a supermarket.

example, Time(6:30...8:00) => Items_bought(milk). Rules like these are called *multidimensional association rules*. The dimensions represent attributes of records of a file or, in terms of relations, columns of rows of a relation, and can be categorical or quantitative. Categorical attributes have a finite set of values that display no ordering relationship. Quantitative attributes are numeric and their values display an ordering relationship, for example, <. Items_bought is an example of a categorical attribute and Transaction_id and Time are quantitative.

One approach to handling a quantitative attribute is to partition its values into nonoverlapping intervals that are assigned labels. This can be done in a static manner based on domain-specific knowledge. For example, a concept hierarchy may group values for Salary into three distinct classes: low income ($0 < \text{Salary} < 29{,}999$), middle income ($30{,}000 < \text{Salary} < 74{,}999$), and high income ($\text{Salary} > 75{,}000$). From here, the typical Apriori-type algorithm or one of its variants can be used for the rule mining since the quantitative attributes now look like categorical attributes. Another approach to partitioning is to group attribute values based on data distribution, for example, equi-depth partitioning, and to assign integer values to each partition. The partitioning at this stage may be relatively fine, that is, a larger number of intervals. Then during the mining process, these partitions may combine with other adjacent partitions if their support is less than some predefined maximum value. An Apriori-type algorithm can be used here as well for the data mining.

Negative Associations. The problem of discovering a negative association is harder than that of discovering a positive association. A negative association is of the following type: *60 percent of customers who buy potato chips do not buy bottled water*. (Here, the 60 percent refers to the confidence for the negative association rule.) In a database with 10,000 items, there are 210,000 possible combinations of items, a majority of which do not appear even once in the database. If the absence of a certain item combination is taken to mean a negative association, then we potentially have millions and millions of negative association rules with RHSs that are of no interest at all. The problem, then, is to find only *interesting* negative rules. In general, we are interested in cases in which two specific sets of items appear very rarely in the same transaction. This poses two problems.

1. For a total item inventory of 10,000 items, the probability of any two being bought together is $(1/10{,}000) * (1/10{,}000) = 10^{-8}$. If we find the actual support for these two occurring together to be zero, that does not represent a significant departure from expectation and hence is not an interesting (negative) association.
2. The other problem is more serious. We are looking for item combinations with very low support, and there are millions and millions with low or even zero support. For example, a data set of 10 million transactions has most of the 2.5 billion pairwise combinations of 10,000 items missing. This would generate billions of useless rules.

Therefore, to make negative association rules interesting, we must use prior knowledge about the itemsets. One approach is to use hierarchies. Suppose we use the hierarchies of soft drinks and chips shown in Figure 28.4.

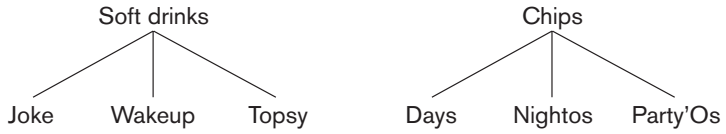


Figure 28.4
Simple hierarchy of soft drinks and chips.

A strong positive association has been shown between soft drinks and chips. If we find a large support for the fact that when customers buy Days chips they predominantly buy Topsy and *not* Joke and *not* Wakeup, that would be interesting because we would normally expect that if there is a strong association between Days and Topsy, there should also be such a strong association between Days and Joke or Days and Wakeup.[5]

In the frozen yogurt and bottled water groupings shown in Figure 28.3, suppose the Reduce versus Healthy-brand division is 80–20 and the Plain and Clear brands division is 60–40 among respective categories. This would give a joint probability of Reduce frozen yogurt being purchased with Plain bottled water as 48 percent among the transactions containing a frozen yogurt and bottled water. If this support, however, is found to be only 20 percent, it would indicate a significant negative association among Reduce yogurt and Plain bottled water; again, that would be interesting.

The problem of finding negative association is important in the above situations given the domain knowledge in the form of item generalization hierarchies (that is, the beverage given and desserts hierarchies shown in Figure 28.3), the existing positive associations (such as between the frozen yogurt and bottled water groups), and the distribution of items (such as the name brands within related groups). The scope of discovery of negative associations is limited in terms of knowing the item hierarchies and distributions. Exponential growth of negative associations remains a challenge.

28.2.7 Additional Considerations for Association Rules

Mining association rules in real-life databases is complicated by the following factors:

- The cardinality of itemsets in most situations is extremely large, and the volume of transactions is very high as well. Some operational databases in retailing and communication industries collect tens of millions of transactions per day.
- Transactions show variability in such factors as geographic location and seasons, making sampling difficult.
- Item classifications exist along multiple dimensions. Hence, driving the discovery process with domain knowledge, particularly for negative rules, is extremely difficult.

[5]For simplicity we are assuming a uniform distribution of transactions among members of a hierarchy.

- Quality of data is variable; significant problems exist with missing, erroneous, conflicting, as well as redundant data in many industries.

28.3 Classification

Classification is the process of learning a model that describes different classes of data. The classes are predetermined. For example, in a banking application, customers who apply for a credit card may be classified as a *poor risk*, *fair risk*, or *good risk*. Hence this type of activity is also called **supervised learning**. Once the model is built, it can be used to classify new data. The first step—learning the model—is accomplished by using a training set of data that has already been classified. Each record in the training data contains an attribute, called the *class* label, which indicates which class the record belongs to. The model that is produced is usually in the form of a decision tree or a set of rules. Some of the important issues with regard to the model and the algorithm that produces the model include the model's ability to predict the correct class of new data, the computational cost associated with the algorithm, and the scalability of the algorithm.

We will examine the approach where our model is in the form of a decision tree. A **decision tree** is simply a graphical representation of the description of each class or, in other words, a representation of the classification rules. A sample decision tree is pictured in Figure 28.5. We see from Figure 28.5 that if a customer is *married* and if salary >= 50K, then they are a good risk for a bank credit card. This is one of the rules that describe the class *good risk*. Traversing the decision tree from the root to each leaf node forms other rules for this class and the two other classes. Algorithm 28.3 shows the procedure for constructing a decision tree from a training data set. Initially, all training samples are at the root of the tree. The samples are partitioned

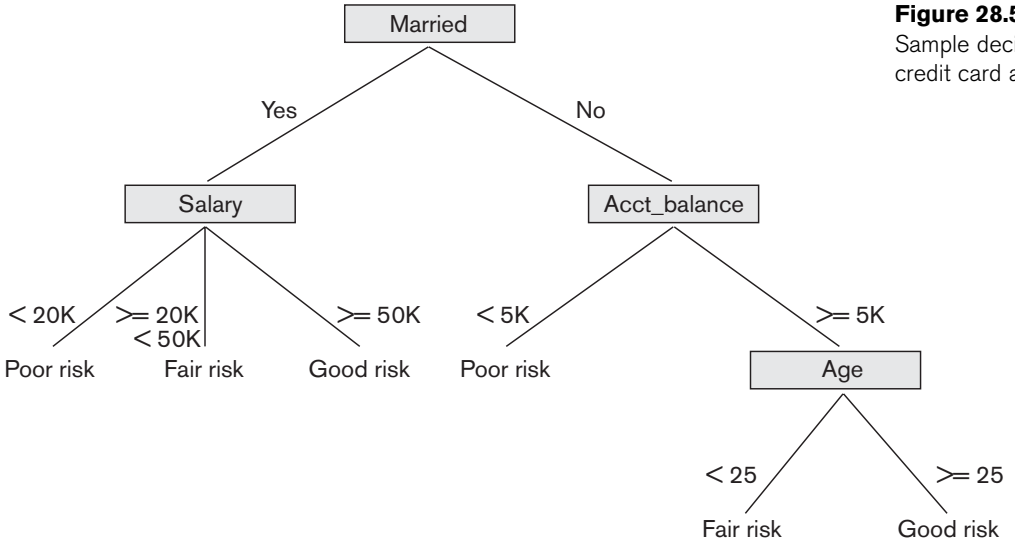


Figure 28.5
Sample decision tree for credit card applications.

recursively based on selected attributes. The attribute used at a node to partition the samples is the one with the best splitting criterion, for example, the one that maximizes the information gain measure.

Algorithm 28.3. Algorithm for Decision Tree Induction

Input: Set of training data records: $R_1, R_2, ..., R_m$ and set of attributes: $A_1, A_2, ..., A_n$

Output: Decision tree

```
procedure Build_tree (records, attributes);
Begin
    create a node N;
    if all records belong to the same class, C then
        return N as a leaf node with class label C;
    if attributes is empty then
        return N as a leaf node with class label C, such that the majority of
        records belong to it;
    select attribute A_i (with the highest information gain) from attributes;
    label node N with A_i;
    for each known value, v_j, of A_i do
        begin
            add a branch from node N for the condition A_i = v_j;
            S_j = subset of records where A_i = v_j;
            if S_j is empty then
                add a leaf, L, with class label C, such that the majority of
                        records belong to it and return L
            else add the node returned by Build_tree(S_j, attributes – A_i);
        end;
End;
```

Before we illustrate Algorithm 28.3, we will explain the **information gain** measure in more detail. The use of **entropy** as the information gain measure is motivated by the goal of minimizing the information needed to classify the sample data in the resulting partitions and thus minimizing the expected number of conditional tests needed to classify a new record. The expected information needed to classify training data of s samples, where the Class attribute has n values ($v_1, ..., v_n$) and s_i is the number of samples belonging to class label v_i, is given by

$$I(S_1, S_2, ..., S_n) = -\sum_{i=1}^{n} p_i \log_2 p_i$$

where p_i is the probability that a random sample belongs to the class with label v_i. An estimate for p_i is s_i/s. Consider an attribute A with values $\{v_1, ..., v_m\}$ used as the test attribute for splitting in the decision tree. Attribute A partitions the samples into the subsets $S_1, ..., S_m$ where samples in each S_j have a value of v_j for attribute A. Each S_j may contain samples that belong to any of the classes. The number of

samples in S_j that belong to class i can be denoted as s_{ij}. The entropy associated with using attribute A as the test attribute is defined as

$$E(A) = \sum_{j=1}^{m} \frac{S_{1j} + \ldots + S_{nj}}{S} \times I\left(S_{1j}, \ldots, S_{nj}\right)$$

$I(s_{1j}, \ldots, s_{nj})$ can be defined using the formulation for $I(s_1, \ldots, s_n)$ with p_i being replaced by p_{ij} where $p_{ij} = s_{ij}/s_j$. Now the information gain by partitioning on attribute A, Gain(A), is defined as $I(s_1, \ldots, s_n) - E(A)$. We can use the sample training data from Figure 28.6 to illustrate the algorithm.

The attribute RID represents the record identifier used for identifying an individual record and is an internal attribute. We use it to identify a particular record in our example. First, we compute the expected information needed to classify the training data of 6 records as $I(s_1, s_2)$ where there are two classes: the first class label value corresponds to *yes* and the second to *no*. So,

$$I(3,3) = -0.5\log_2 0.5 - 0.5\log_2 0.5 = 1.$$

Now, we compute the entropy for each of the four attributes as shown below. For Married = yes, we have $s_{11} = 2$, $s_{21} = 1$ and $I(s_{11}, s_{21}) = 0.92$. For Married = no, we have $s_{12} = 1$, $s_{22} = 2$ and $I(s_{12}, s_{22}) = 0.92$. So, the expected information needed to classify a sample using attribute Married as the partitioning attribute is

$$E(\text{Married}) = 3/6\ I(s_{11}, s_{21}) + 3/6\ \text{I}(s_{12}, s_{22}) = 0.92.$$

The gain in information, Gain(Married), would be $1 - 0.92 = 0.08$. If we follow similar steps for computing the gain with respect to the other three attributes we end up with

E(Salary) = 0.33	and	Gain(Salary) = 0.67
E(Acct_balance) = 0.92	and	Gain(Acct_balance) = 0.08
E(Age) = 0.54	and	Gain(Age) = 0.46

Since the greatest gain occurs for attribute Salary, it is chosen as the partitioning attribute. The root of the tree is created with label *Salary* and has three branches, one for each value of Salary. For two of the three values, that is, <20K and >=50K, all the samples that are partitioned accordingly (records with RIDs 4 and 5 for <20K

RID	Married	Salary	Acct_balance	Age	Loanworthy
1	no	>=50K	<5K	>=25	yes
2	yes	>=50K	>=5K	>=25	yes
3	yes	20K. . .50K	<5K	<25	no
4	no	<20K	>=5K	<25	no
5	no	<20K	<5K	>=25	no
6	yes	20K. . .50K	>=5K	>=25	yes

Figure 28.6
Sample training data for classification algorithm.

and records with RIDs 1 and 2 for >=50K) fall within the same class *loanworthy no* and *loanworthy yes* respectively for those two values. So we create a leaf node for each. The only branch that needs to be expanded is for the value 20K...50K with two samples, records with RIDs 3 and 6 in the training data. Continuing the process using these two records, we find that Gain(Married) is 0, Gain(Acct_balance) is 1, and Gain(Age) is 1.

We can choose either Age or Acct_balance since they both have the largest gain. Let us choose Age as the partitioning attribute. We add a node with label *Age* that has two branches, less than 25, and greater or equal to 25. Each branch partitions the remaining sample data such that one sample record belongs to each branch and hence one class. Two leaf nodes are created and we are finished. The final decision tree is pictured in Figure 28.7.

28.4 Clustering

The previous data mining task of classification deals with partitioning data based on using a preclassified training sample. However, it is often useful to partition data without having a training sample; this is also known as **unsupervised learning**. For example, in business, it may be important to determine groups of customers who have similar buying patterns, or in medicine, it may be important to determine groups of patients who show similar reactions to prescribed drugs. The goal of clustering is to place records into groups, such that records in a group are similar to each other and dissimilar to records in other groups. The groups are usually *disjoint*.

An important facet of clustering is the similarity function that is used. When the data is numeric, a similarity function based on distance is typically used. For example, the Euclidean distance can be used to measure similarity. Consider two n-dimensional data points (records) r_j and r_k. We can consider the value for the ith dimension as r_{ji} and r_{ki} for the two records. The Euclidean distance between points r_j and r_k in n-dimensional space is calculated as:

$$\text{Distance}(r_j, r_k) = \sqrt{|r_{j1} - r_{k1}|^2 + |r_{j2} - r_{k2}|^2 + \ldots + |r_{jn} - r_{kn}|^2}$$

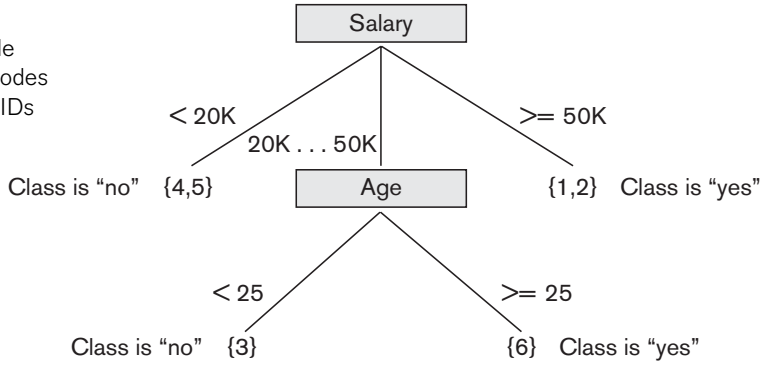


Figure 28.7
Decision tree based on sample training data where the leaf nodes are represented by a set of RIDs of the partitioned records.

The smaller the distance between two points, the greater is the similarity as we think of them. A classic clustering algorithm is the *k*-Means algorithm, Algorithm 28.4.

Algorithm 28.4. *k*-Means Clustering Algorithm

Input: a database D, of m records, $r_1, ..., r_m$ and a desired number of clusters k

Output: set of k clusters that minimizes the squared error criterion

Begin
randomly choose k records as the centroids for the k clusters;
repeat
assign each record, r_i, to a cluster such that the distance between r_i and the cluster centroid (mean) is the smallest among the k clusters;
recalculate the centroid (mean) for each cluster based on the records assigned to the cluster;
until no change;
End;

The algorithm begins by randomly choosing k records to represent the centroids (means), $m_1, ..., m_k$, of the clusters, $C_1, ..., C_k$. All the records are placed in a given cluster based on the distance between the record and the cluster mean. If the distance between m_i and record r_j is the smallest among all cluster means, then record r_j is placed in cluster C_i. Once all records have been initially placed in a cluster, the mean for each cluster is recomputed. Then the process repeats, by examining each record again and placing it in the cluster whose mean is closest. Several iterations may be needed, but the algorithm will converge, although it may terminate at a local optimum. The terminating condition is usually the squared-error criterion. For clusters $C_1, ..., C_k$ with means $m_1, ..., m_k$, the error is defined as:

$$\text{Error} = \sum_{i=1}^{k} \sum_{\forall r_j \in C_i} \text{Distance}(r_j, m_i)^2$$

We will examine how Algorithm 28.4 works with the (two-dimensional) records in Figure 28.8. Assume that the number of desired clusters k is 2. Let the algorithm choose records with RID 3 for cluster C_1 and RID 6 for cluster C_2 as the initial cluster centroids. The remaining records will be assigned to one of those clusters during the

RID	Age	Years_of_service
1	30	5
2	50	25
3	50	15
4	25	5
5	30	10
6	55	25

Figure 28.8
Sample 2-dimensional records for clustering example (the RID column is not considered).

first iteration of the repeat loop. The record with RID 1 has a distance from C_1 of 22.4 and a distance from C_2 of 32.0, so it joins cluster C_1. The record with RID 2 has a distance from C_1 of 10.0 and a distance from C_2 of 5.0, so it joins cluster C_2. The record with RID 4 has a distance from C_1 of 25.5 and a distance from C_2 of 36.6, so it joins cluster C_1. The record with RID 5 has a distance from C_1 of 20.6 and a distance from C_2 of 29.2, so it joins cluster C_1. Now, the new means (centroids) for the two clusters are computed. The mean for a cluster, C_i, with n records of m dimensions is the vector:

$$\bar{C}_i = \left(\frac{1}{n} \sum_{\forall r_j \in C_i} r_{ji}, \ldots, \frac{1}{n} \sum_{\forall r_j \in C_i} r_{jm} \right)$$

The new mean for C_1 is (33.75, 8.75) and the new mean for C_2 is (52.5, 25). A second iteration proceeds and the six records are placed into the two clusters as follows: records with RIDs 1, 4, 5 are placed in C_1 and records with RIDs 2, 3, 6 are placed in C_2. The mean for C_1 and C_2 is recomputed as (28.3, 6.7) and (51.7, 21.7), respectively. In the next iteration, all records stay in their previous clusters and the algorithm terminates.

Traditionally, clustering algorithms assume that the entire data set fits in main memory. More recently, researchers have developed algorithms that are efficient and are scalable for very large databases. One such algorithm is called BIRCH. BIRCH is a hybrid approach that uses both a hierarchical clustering approach, which builds a tree representation of the data, as well as additional clustering methods, which are applied to the leaf nodes of the tree. Two input parameters are used by the BIRCH algorithm. One specifies the amount of available main memory and the other is an initial threshold for the radius of any cluster. Main memory is used to store descriptive cluster information such as the center (mean) of a cluster and the radius of the cluster (clusters are assumed to be spherical in shape). The radius threshold affects the number of clusters that are produced. For example, if the radius threshold value is large, then few clusters of many records will be formed. The algorithm tries to maintain the number of clusters such that their radius is below the radius threshold. If available memory is insufficient, then the radius threshold is increased.

The BIRCH algorithm reads the data records sequentially and inserts them into an in-memory tree structure, which tries to preserve the clustering structure of the data. The records are inserted into the appropriate leaf nodes (potential clusters) based on the distance between the record and the cluster center. The leaf node where the insertion happens may have to split, depending upon the updated center and radius of the cluster and the radius threshold parameter. Additionally, when splitting, extra cluster information is stored, and if memory becomes insufficient, then the radius threshold will be increased. Increasing the radius threshold may actually produce a side effect of reducing the number of clusters since some nodes may be merged.

Overall, BIRCH is an efficient clustering method with a linear computational complexity in terms of the number of records to be clustered.

28.5 Approaches to Other Data Mining Problems

28.5.1 Discovery of Sequential Patterns

The discovery of sequential patterns is based on the concept of a sequence of itemsets. We assume that transactions such as the supermarket-basket transactions we discussed previously are ordered by time of purchase. That ordering yields a sequence of itemsets. For example, {milk, bread, juice}, {bread, eggs}, {cookies, milk, coffee} may be such a **sequence of itemsets** based on three visits by the same customer to the store. The **support** for a sequence S of itemsets is the percentage of the given set U of sequences of which S is a subsequence. In this example, {milk, bread, juice} {bread, eggs} and {bread, eggs} {cookies, milk, coffee} are considered **subsequences**. The problem of identifying sequential patterns, then, is to find all subsequences from the given sets of sequences that have a user-defined minimum support. The sequence S_1, S_2, S_3, ... is a **predictor** of the fact that a customer who buys itemset S_1 is likely to buy itemset S_2 and then S_3, and so on. This prediction is based on the frequency (support) of this sequence in the past. Various algorithms have been investigated for sequence detection.

28.5.2 Discovery of Patterns in Time Series

Time series are sequences of events; each event may be a given fixed type of a transaction. For example, the closing price of a stock or a fund is an event that occurs every weekday for each stock and fund. The sequence of these values per stock or fund constitutes a time series. For a time series, one may look for a variety of patterns by analyzing sequences and subsequences as we did above. For example, we might find the period during which the stock rose or held steady for n days, or we might find the longest period over which the stock had a fluctuation of no more than 1 percent over the previous closing price, or we might find the quarter during which the stock had the most percentage gain or percentage loss. Time series may be compared by establishing measures of similarity to identify companies whose stocks behave in a similar fashion. Analysis and mining of time series is an extended functionality of temporal data management (see Chapter 26).

28.5.3 Regression

Regression is a special application of the classification rule. If a classification rule is regarded as a function over the variables that maps these variables into a target class variable, the rule is called a **regression rule**. A general application of regression occurs when, instead of mapping a tuple of data from a relation to a specific class, the value of a variable is predicted based on that tuple. For example, consider a relation

LAB_TESTS (patient ID, test 1, test 2, ..., test n)

which contains values that are results from a series of n tests for one patient. The target variable that we wish to predict is P, the probability of survival of the patient. Then the rule for regression takes the form:

(test 1 in range$_1$) and (test 2 in range$_2$) and ... (test n in range$_n$) $\Rightarrow P = x$, or $x < P \leq y$

The choice depends on whether we can predict a unique value of P or a range of values for P. If we regard P as a function:

$P = f$ (test 1, test 2, ..., test n)

the function is called a **regression function** to predict P. In general, if the function appears as

$Y = f(X_1, X_2, ..., X_n)$,

and f is linear in the domain variables x_i, the process of deriving f from a given set of tuples for $<X_1, X_2, ..., X_n, Y>$ is called **linear regression**. Linear regression is a commonly used statistical technique for fitting a set of observations or points in n dimensions with the target variable y.

Regression analysis is a very common tool for analysis of data in many research domains. The discovery of the function to predict the target variable is equivalent to a data mining operation.

28.5.4 Neural Networks

A **neural network** is a technique derived from artificial intelligence research that uses generalized regression and provides an iterative method to carry it out. Neural networks use the curve-fitting approach to infer a function from a set of samples. This technique provides a *learning approach*; it is driven by a test sample that is used for the initial inference and learning. With this kind of learning method, responses to new inputs may be able to be interpolated from the known samples. This interpolation, however, depends on the world model (internal representation of the problem domain) developed by the learning method.

Neural networks can be broadly classified into two categories: supervised and unsupervised networks. Adaptive methods that attempt to reduce the output error are **supervised learning** methods, whereas those that develop internal representations without sample outputs are called **unsupervised learning** methods.

Neural networks self-adapt; that is, they learn from information about a specific problem. They perform well on classification tasks and are therefore useful in data mining. Yet, they are not without problems. Although they learn, they do not provide a good representation of *what* they have learned. Their outputs are highly quantitative and not easy to understand. As another limitation, the internal representations developed by neural networks are not unique. Also, in general, neural networks have trouble modeling time series data. Despite these shortcomings, they are popular and frequently used by several commercial vendors.

28.5.5 Genetic Algorithms

Genetic algorithms (GAs) are a class of randomized search procedures capable of adaptive and robust search over a wide range of search space topologies. Modeled after the adaptive emergence of biological species from evolutionary mechanisms, and introduced by Holland,[6] GAs have been successfully applied in such diverse fields as image analysis, scheduling, and engineering design.

Genetic algorithms extend the idea from human genetics of the four-letter alphabet (based on the A,C,T,G nucleotides) of the human DNA code. The construction of a genetic algorithm involves devising an alphabet that encodes the solutions to the decision problem in terms of strings of that alphabet. Strings are equivalent to individuals. A fitness function defines which solutions can survive and which cannot. The ways in which solutions can be combined are patterned after the cross-over operation of cutting and combining strings from a father and a mother. An initial population of a well-varied population is provided, and a game of evolution is played in which mutations occur among strings. They combine to produce a new generation of individuals; the fittest individuals survive and mutate until a family of successful solutions develops.

The solutions produced by GAs are distinguished from most other search techniques by the following characteristics:

- A GA search uses a set of solutions during each generation rather than a single solution.
- The search in the string-space represents a much larger parallel search in the space of encoded solutions.
- The memory of the search done is represented solely by the set of solutions available for a generation.
- A genetic algorithm is a randomized algorithm since search mechanisms use probabilistic operators.
- While progressing from one generation to the next, a GA finds near-optimal balance between knowledge acquisition and exploitation by manipulating encoded solutions.

Genetic algorithms are used for problem solving and clustering problems. Their ability to solve problems in parallel provides a powerful tool for data mining. The drawbacks of GAs include the large overproduction of individual solutions, the random character of the searching process, and the high demand on computer processing. In general, substantial computing power is required to achieve anything of significance with genetic algorithms.

[6]Holland's seminal work (1975) entitled *Adaptation in Natural and Artificial Systems* introduced the idea of genetic algorithms.

28.6 Applications of Data Mining

Data mining technologies can be applied to a large variety of decision-making contexts in business. In particular, areas of significant payoffs are expected to include the following:

- **Marketing.** Applications include analysis of consumer behavior based on buying patterns; determination of marketing strategies including advertising, store location, and targeted mailing; segmentation of customers, stores, or products; and design of catalogs, store layouts, and advertising campaigns.
- **Finance.** Applications include analysis of creditworthiness of clients, segmentation of account receivables, performance analysis of finance investments like stocks, bonds, and mutual funds; evaluation of financing options; and fraud detection.
- **Manufacturing.** Applications involve optimization of resources like machines, manpower, and materials; and optimal design of manufacturing processes, shop-floor layouts, and product design, such as for automobiles based on customer requirements.
- **Health Care.** Applications include discovery of patterns in radiological images, analysis of microarray (gene-chip) experimental data to cluster genes and to relate to symptoms or diseases, analysis of side effects of drugs and effectiveness of certain treatments, optimization of processes within a hospital, and the relationship of patient wellness data with doctor qualifications.

28.7 Commercial Data Mining Tools

Currently, commercial data mining tools use several common techniques to extract knowledge. These include association rules, clustering, neural networks, sequencing, and statistical analysis. We discussed these earlier. Also used are decision trees, which are a representation of the rules used in classification or clustering, and statistical analyses, which may include regression and many other techniques. Other commercial products use advanced techniques such as genetic algorithms, case-based reasoning, Bayesian networks, nonlinear regression, combinatorial optimization, pattern matching, and fuzzy logic. In this chapter we have already discussed some of these.

Most data mining tools use the ODBC (Open Database Connectivity) interface. ODBC is an industry standard that works with databases; it enables access to data in most of the popular database programs such as Access, dBASE, Informix, Oracle, and SQL Server. Some of these software packages provide interfaces to specific database programs; the most common are Oracle, Access, and SQL Server. Most of the tools work in the Microsoft Windows environment and a few work in the UNIX operating system. The trend is for all products to operate under the Microsoft Windows environment. One tool, Data Surveyor, mentions ODMG compliance; see Chapter 11 where we discuss the ODMG object-oriented standard.

In general, these programs perform sequential processing in a single machine. Many of these products work in the client-server mode. Some products incorporate parallel processing in parallel computer architectures and work as a part of online analytical processing (OLAP) tools.

28.7.1 User Interface

Most of the tools run in a graphical user interface (GUI) environment. Some products include sophisticated visualization techniques to view data and rules (for example, SGI's MineSet), and are even able to manipulate data this way interactively. Text interfaces are rare and are more common in tools available for UNIX, such as IBM's Intelligent Miner.

28.7.2 Application Programming Interface

Usually, the application programming interface (API) is an optional tool. Most products do not permit using their internal functions. However, some of them allow the application programmer to reuse their code. The most common interfaces are C libraries and Dynamic Link Libraries (DLLs). Some tools include proprietary database command languages.

In Table 28.1 we list 11 representative data mining tools. To date, there are almost one hundred commercial data mining products available worldwide. Non-U.S. products include Data Surveyor from the Netherlands and PolyAnalyst from Russia.

28.7.3 Future Directions

Data mining tools are continually evolving, building on ideas from the latest scientific research. Many of these tools incorporate the latest algorithms taken from artificial intelligence (AI), statistics, and optimization.

Currently, fast processing is done using modern database techniques—such as distributed processing—in client-server architectures, in parallel databases, and in data warehousing. For the future, the trend is toward developing Internet capabilities more fully. Additionally, hybrid approaches will become commonplace, and processing will be done using all resources available. Processing will take advantage of both parallel and distributed computing environments. This shift is especially important because modern databases contain very large amounts of information. Not only are multimedia databases growing, but also image storage and retrieval are slow operations. Also, the cost of secondary storage is decreasing, so massive information storage will be feasible, even for small companies. Thus, data mining programs will have to deal with larger sets of data of more companies.

Most of data mining software will use the ODBC standard to extract data from business databases; proprietary input formats can be expected to disappear. There is a definite need to include nonstandard data, including images and other multimedia data, as source data for data mining.

Table 28.1 Some Representative Data Mining Tools

Company	Product	Technique	Platform	Interface*
AcknoSoft	Kate	Decision trees, Case-based reasoning	Windows UNIX	Microsoft Access
Angoss	Knowledge SEEKER	Decision trees, Statistics	Windows	ODBC
Business Objects	Business Miner	Neural nets, Machine learning	Windows	ODBC
CrossZ	QueryObject	Statistical analysis, Optimization algorithm	Windows MVS UNIX	ODBC
Data Distilleries	Data Surveyor	Comprehensive; can mix different types of data mining	UNIX	ODBC ODMG-compliant
DBMiner Technology Inc.	DBMiner	OLAP analysis, Associations, Classification, Clustering algorithms	Windows	Microsoft 7.0 OLAP
IBM	Intelligent Miner	Classification, Association rules, Predictive models	UNIX (AIX)	IBM DB2
Megaputer Intelligence	PolyAnalyst	Symbolic knowledge acquisition, Evolutionary programming	Windows OS/2	ODBC Oracle DB2
NCR	Management Discovery Tool (MDT)	Association rules	Windows	ODBC
Purple Insight	MineSet	Decision trees, Association rules	UNIX (Irix)	Oracle Sybase Informix
SAS	Enterprise Miner	Decision trees, Association rules, Neural nets, Regression, Clustering	UNIX (Solaris) Windows Macintosh	ODBC Oracle AS/400

*ODBC: Open Data Base Connectivity

ODMG: Object Data Management Group

28.8 Summary

In this chapter we surveyed the important discipline of data mining, which uses database technology to discover additional knowledge or patterns in the data. We gave an illustrative example of knowledge discovery in databases, which has a wider scope than data mining. For data mining, among the various techniques, we focused on the details of association rule mining, classification, and clustering. We presented algorithms in each of these areas and illustrated with examples of how those algorithms work.

A variety of other techniques, including the AI-based neural networks and genetic algorithms, were also briefly discussed. Active research is ongoing in data mining and we have outlined some of the expected research directions. In the future database technology products market, a great deal of data mining activity is expected. We summarized 11 out of nearly one hundred data mining tools available; future research is expected to extend the number and functionality significantly.

Review Questions

28.1. What are the different phases of the knowledge discovery from databases? Describe a complete application scenario in which new knowledge may be mined from an existing database of transactions.

28.2. What are the goals or tasks that data mining attempts to facilitate?

28.3. What are the five types of knowledge produced from data mining?

28.4. What are association rules as a type of knowledge? Give a definition of support and confidence and use them to define an association rule.

28.5. What is the downward closure property? How does it aid in developing an efficient algorithm for finding association rules, that is, with regard to finding large itemsets?

28.6. What was the motivating factor for the development of the FP-tree algorithm for association rule mining?

28.7. Describe an association rule among hierarchies with an example.

28.8. What is a negative association rule in the context of the hierarchy in Figure 28.3?

28.9. What are the difficulties of mining association rules from large databases?

28.10. What are classification rules and how are decision trees related to them?

28.11. What is entropy and how is it used in building decision trees?

28.12. How does clustering differ from classification?

28.13. Describe neural networks and genetic algorithms as techniques for data mining. What are the main difficulties in using these techniques?

Exercises

28.14. Apply the Apriori algorithm to the following data set.

Trans_id	Items_purchased
101	milk, bread, eggs
102	milk, juice
103	juice, butter
104	milk, bread, eggs
105	coffee, eggs
106	coffee
107	coffee, juice
108	milk, bread, cookies, eggs
109	cookies, butter
110	milk, bread

The set of items is {milk, bread, cookies, eggs, butter, coffee, juice}. Use 0.2 for the minimum support value.

28.15. Show two rules that have a confidence of 0.7 or greater for an itemset containing three items from Exercise 28.14.

28.16. For the Partition algorithm, prove that any frequent itemset in the database must appear as a local frequent itemset in at least one partition.

28.17. Show the FP-tree that would be made for the data from Exercise 28.14.

28.18. Apply the FP-Growth algorithm to the FP-tree from Exercise 28.17 and show the frequent itemsets.

28.19. Apply the classification algorithm to the following set of data records. The class attribute is Repeat_customer.

RID	Age	City	Gender	Education	Repeat_customer
101	20...30	NY	F	college	YES
102	20...30	SF	M	graduate	YES
103	31...40	NY	F	college	YES
104	51...60	NY	F	college	NO
105	31...40	LA	M	high school	NO
106	41...50	NY	F	college	YES
107	41...50	NY	F	graduate	YES
108	20...30	LA	M	college	YES
109	20...30	NY	F	high school	NO
110	20...30	NY	F	college	YES

28.20. Consider the following set of two-dimensional records:

RID	Dimension1	Dimension2
1	8	4
2	5	4
3	2	4
4	2	6
5	2	8
6	8	6

Also consider two different clustering schemes: (1) where $Cluster_1$ contains records {1,2,3} and $Cluster_2$ contains records {4,5,6} and (2) where $Cluster_1$ contains records {1,6} and $Cluster_2$ contains records {2,3,4,5}. Which scheme is better and why?

28.21. Use the *k*-Means algorithm to cluster the data from Exercise 28.20. We can use a value of 3 for *K* and we can assume that the records with RIDs 1, 3, and 5 are used for the initial cluster centroids (means).

28.22. The *k*-Means algorithm uses a similarity metric of distance between a record and a cluster centroid. If the attributes of the records are not quantitative but categorical in nature, such as Income_level with values {low, medium, high} or Married with values {Yes, No} or State_of_residence with values {Alabama, Alaska, ..., Wyoming}, then the distance metric is not meaningful. Define a more suitable similarity metric that can be used for clustering data records that contain categorical data.

Selected Bibliography

Literature on data mining comes from several fields, including statistics, mathematical optimization, machine learning, and artificial intelligence. Chen et al. (1996) give a good summary of the database perspective on data mining. The book by Han and Kamber (2001) is an excellent text, describing in detail the different algorithms and techniques used in the data mining area. Work at IBM Almaden research has produced a large number of early concepts and algorithms as well as results from some performance studies. Agrawal et al. (1993) report the first major study on association rules. Their Apriori algorithm for market basket data in Agrawal and Srikant (1994) is improved by using partitioning in Savasere et al. (1995); Toivonen (1996) proposes sampling as a way to reduce the processing effort. Cheung et al. (1996) extends the partitioning to distributed environments; Lin and Dunham (1998) propose techniques to overcome problems with data skew. Agrawal et al. (1993b) discuss the performance perspective on association rules. Mannila et al. (1994), Park et al. (1995), and Amir et al. (1997) present additional efficient algorithms related to association rules. Han et al. (2000) present the FP-tree algorithm

discussed in this chapter. Srikant and Agrawal(1995) proposes mining generalized rules. Savasere et al. (1998) present the first approach to mining negative associations. Agrawal et al. (1996) describe the Quest system at IBM. Sarawagi et al. (1998) describe an implementation where association rules are integrated with a relational database management system. Piatesky-Shapiro and Frawley (1992) have contributed papers from a wide range of topics related to knowledge discovery. Zhang et al. (1996) present the BIRCH algorithm for clustering large databases. Information about decision tree learning and the classification algorithm presented in this chapter can be found in Mitchell (1997).

Adriaans and Zantinge (1996), Fayyad et al. (1997), and Weiss and Indurkhya (1998) are books devoted to the different aspects of data mining and its use in prediction. The idea of genetic algorithms was proposed by Holland (1975); a good survey of genetic algorithms appears in Srinivas and Patnaik (1994). Neural networks have a vast literature; a comprehensive introduction is available in Lippman (1987).

Tan et al. (2006) provides a comprehensive introduction to data mining and has a detailed set of references. Readers are also advised to consult proceedings of two prominent annual conferences in data mining: the Knowledge Discovery and Data Mining Conference (KDD), which has been running since 1995, and the SIAM International Conference on Data Mining (SDM), which has been running since 2001. Links to past conferences may be found at http://dblp.uni-trier.de.

chapter 29

Overview of Data Warehousing and OLAP

The increasing processing power and sophistication of analytical tools and techniques have resulted in the development of what are known as data warehouses. These data warehouses provide storage, functionality, and responsiveness to queries beyond the capabilities of transaction-oriented databases. Accompanying this ever-increasing power is a great demand to improve the data access performance of databases. As we have seen throughout this book, traditional databases balance the requirement of data access with the need to ensure data integrity. In modern organizations, users of data are often completely removed from the data sources. Many people only need read-access to data, but still need fast access to a larger volume of data than can conveniently be downloaded to the desktop. Often such data comes from multiple databases. Because many of the analyses performed are recurrent and predictable, software vendors and systems support staff are designing systems to support these functions. Presently there is a great need to provide decision makers from middle management upward with information at the correct level of detail to support decision making. *Data warehousing, online analytical processing* (OLAP), and *data mining* provide this functionality. We gave an introduction to data mining techniques in Chapter 28. In this chapter we give a broad overview of data warehousing and OLAP technologies.

29.1 Introduction, Definitions, and Terminology

In Chapter 1 we defined a *database* as a collection of related data and a *database system* as a database and database software together. A data warehouse is also a collection of information as well as a supporting system. However, a clear distinction

exists. Traditional databases are transactional (relational, object-oriented, network, or hierarchical). *Data warehouses* have the distinguishing characteristic that they are mainly intended for decision-support applications. They are optimized for data retrieval, not routine transaction processing.

Because data warehouses have been developed in numerous organizations to meet particular needs, there is no single, canonical definition of the term data warehouse. Professional magazine articles and books in the popular press have elaborated on the meaning in a variety of ways. Vendors have capitalized on the popularity of the term to help market a variety of related products, and consultants have provided a large variety of services, all under the data warehousing banner. However, data warehouses are quite distinct from traditional databases in their structure, functioning, performance, and purpose.

W. H. Inmon[1] characterized a **data warehouse** as *a subject-oriented, integrated, nonvolatile, time-variant collection of data in support of management's decisions*. Data warehouses provide access to data for complex analysis, knowledge discovery, and decision making. They support high-performance demands on an organization's data and information. Several types of applications—OLAP, DSS, and data mining applications—are supported. We define each of these next.

OLAP (online analytical processing) is a term used to describe the analysis of complex data from the data warehouse. In the hands of skilled knowledge workers, OLAP tools use distributed computing capabilities for analyses that require more storage and processing power than can be economically and efficiently located on an individual desktop.

DSS (decision-support systems), also known as **EIS—executive information systems**; not to be confused with enterprise integration systems—support an organization's leading decision makers with higher-level data for complex and important decisions. Data mining (which we discussed in Chapter 28) is used for *knowledge discovery*, the process of searching data for unanticipated new knowledge.

Traditional databases support **online transaction processing (OLTP)**, which includes insertions, updates, and deletions, while also supporting information query requirements. Traditional relational databases are optimized to process queries that may touch a small part of the database and transactions that deal with insertions or updates of a few tuples per relation to process. Thus, they cannot be optimized for OLAP, DSS, or data mining. By contrast, data warehouses are designed precisely to support efficient extraction, processing, and presentation for analytic and decision-making purposes. In comparison to traditional databases, data warehouses generally contain very large amounts of data from multiple sources that may include databases from different data models and sometimes files acquired from independent systems and platforms.

[1]Inmon (1992) is credited with initially using the term *warehouse*. The latest edition of his work is Inmon (2005).

29.2 Characteristics of Data Warehouses

To discuss data warehouses and distinguish them from transactional databases calls for an appropriate data model. The multidimensional data model (explained in more detail in Section 29.3) is a good fit for OLAP and decision-support technologies. In contrast to multidatabases, which provide access to disjoint and usually heterogeneous databases, a data warehouse is frequently a store of integrated data from multiple sources, processed for storage in a multidimensional model. Unlike most transactional databases, data warehouses typically support time-series and trend analysis, both of which require more historical data than is generally maintained in transactional databases.

Compared with transactional databases, data warehouses are nonvolatile. This means that information in the data warehouse changes far less often and may be regarded as non–real-time with periodic updating. In transactional systems, transactions are the unit and are the agent of change to the database; by contrast, data warehouse information is much more coarse-grained and is refreshed according to a careful choice of refresh policy, usually incremental. Warehouse updates are handled by the warehouse's acquisition component that provides all required preprocessing.

We can also describe data warehousing more generally as *a collection of decision support technologies, aimed at enabling the knowledge worker (executive, manager, analyst) to make better and faster decisions*.[2] Figure 29.1 gives an overview of the conceptual structure of a data warehouse. It shows the entire data warehousing process, which includes possible cleaning and reformatting of data before loading it into the warehouse. This process is handled by tools known as ETL (extraction, transformation, and loading) tools. At the back end of the process, OLAP, data mining, and DSS may generate new relevant information such as rules; this information is shown in the figure going back into the warehouse. The figure also shows that data sources may include files.

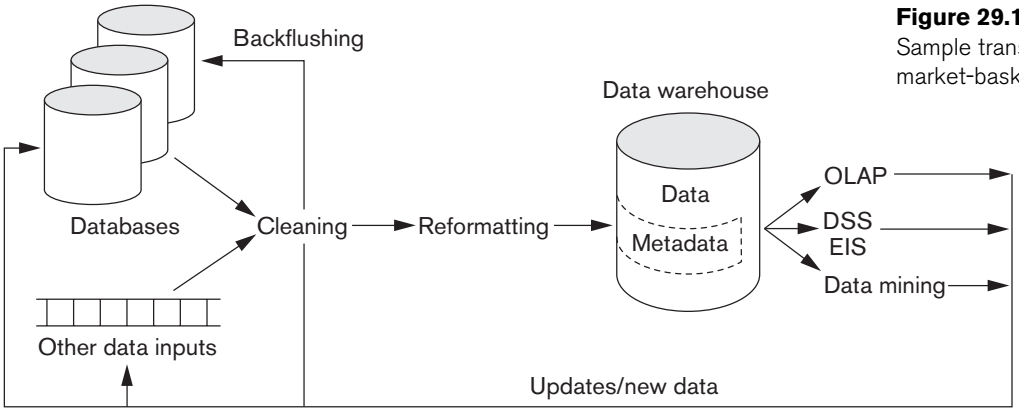


Figure 29.1
Sample transactions in market-basket model.

[2]Chaudhuri and Dayal (1997) provide an excellent tutorial on the topic, with this as a starting definition.

Data warehouses have the following distinctive characteristics:[3]

- Multidimensional conceptual view
- Generic dimensionality
- Unlimited dimensions and aggregation levels
- Unrestricted cross-dimensional operations
- Dynamic sparse matrix handling
- Client-server architecture
- Multiuser support
- Accessibility
- Transparency
- Intuitive data manipulation
- Consistent reporting performance
- Flexible reporting

Because they encompass large volumes of data, data warehouses are generally an order of magnitude (sometimes two orders of magnitude) larger than the source databases. The sheer volume of data (likely to be in terabytes or even petabytes) is an issue that has been dealt with through enterprise-wide data warehouses, virtual data warehouses, and data marts:

- **Enterprise-wide data warehouses** are huge projects requiring massive investment of time and resources.
- **Virtual data warehouses** provide views of operational databases that are materialized for efficient access.
- **Data marts** generally are targeted to a subset of the organization, such as a department, and are more tightly focused.

29.3 Data Modeling for Data Warehouses

Multidimensional models take advantage of inherent relationships in data to populate data in multidimensional matrices called *data cubes*. (These may be called *hypercubes* if they have more than three dimensions.) For data that lends itself to dimensional formatting, query performance in multidimensional matrices can be much better than in the relational data model. Three examples of dimensions in a corporate data warehouse are the corporation's fiscal periods, products, and regions.

A standard spreadsheet is a two-dimensional matrix. One example would be a spreadsheet of regional sales by product for a particular time period. Products could be shown as rows, with sales revenues for each region comprising the columns. (Figure 29.2 shows this two-dimensional organization.) Adding a time dimension,

[3]Codd and Salley (1993) coined the term OLAP and mentioned these characteristics. We have reordered their original list.

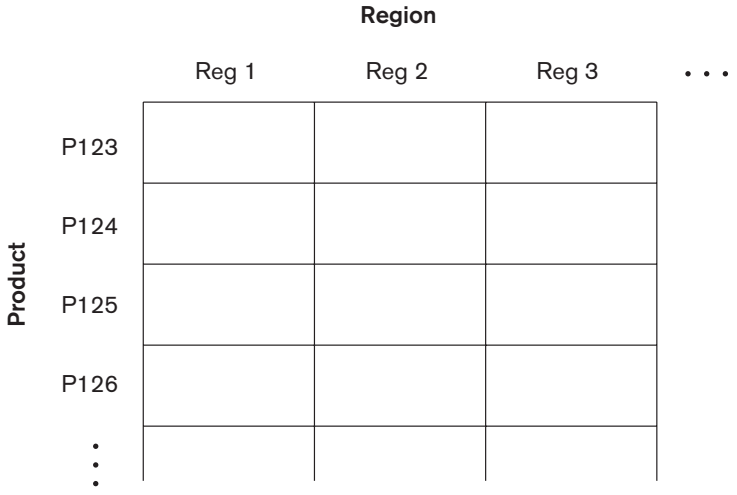


Figure 29.2
A two-dimensional matrix model.

such as an organization's fiscal quarters, would produce a three-dimensional matrix, which could be represented using a data cube.

Figure 29.3 shows a three-dimensional data cube that organizes product sales data by fiscal quarters and sales regions. Each cell could contain data for a specific product,

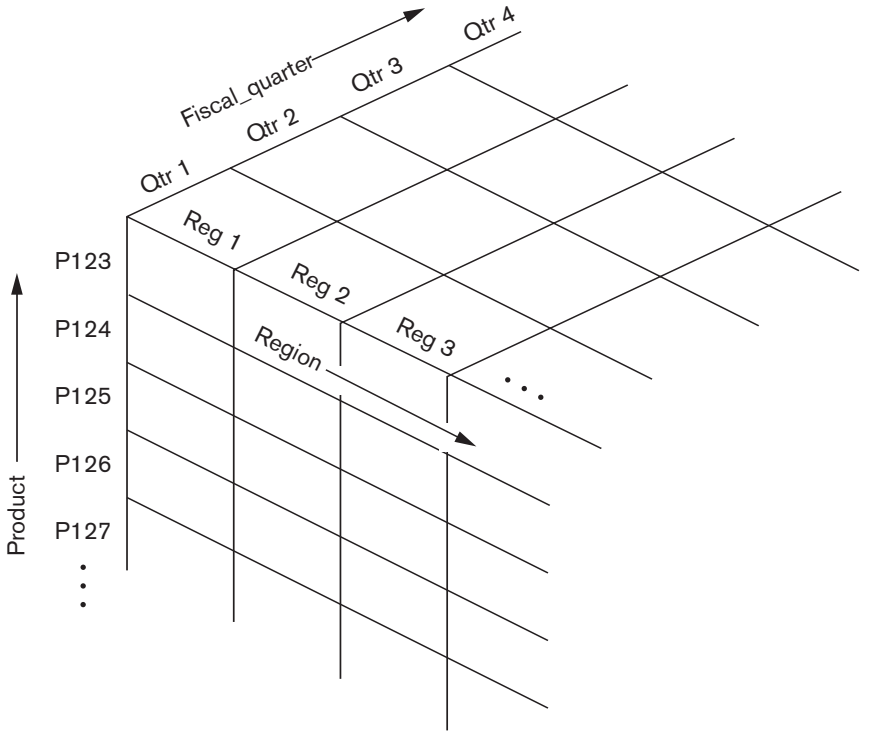


Figure 29.3
A three-dimensional data cube model.

specific fiscal quarter, and specific region. By including additional dimensions, a data hypercube could be produced, although more than three dimensions cannot be easily visualized or graphically presented. The data can be queried directly in any combination of dimensions, bypassing complex database queries. Tools exist for viewing data according to the user's choice of dimensions.

Changing from one-dimensional hierarchy (orientation) to another is easily accomplished in a data cube with a technique called **pivoting** (also called *rotation*). In this technique the data cube can be thought of as rotating to show a different orientation of the axes. For example, you might pivot the data cube to show regional sales revenues as rows, the fiscal quarter revenue totals as columns, and the company's products in the third dimension (Figure 29.4). Hence, this technique is equivalent to having a regional sales table for each product separately, where each table shows quarterly sales for that product region by region.

Multidimensional models lend themselves readily to hierarchical views in what is known as roll-up display and drill-down display. A **roll-up display** moves up the hierarchy, grouping into larger units along a dimension (for example, summing weekly data by quarter or by year). Figure 29.5 shows a roll-up display that moves from individual products to a coarser-grain of product categories. Shown in Figure 29.6, a **drill-down display** provides the opposite capability, furnishing a finer-grained view, perhaps disaggregating country sales by region and then regional sales by subregion and also breaking up products by styles.

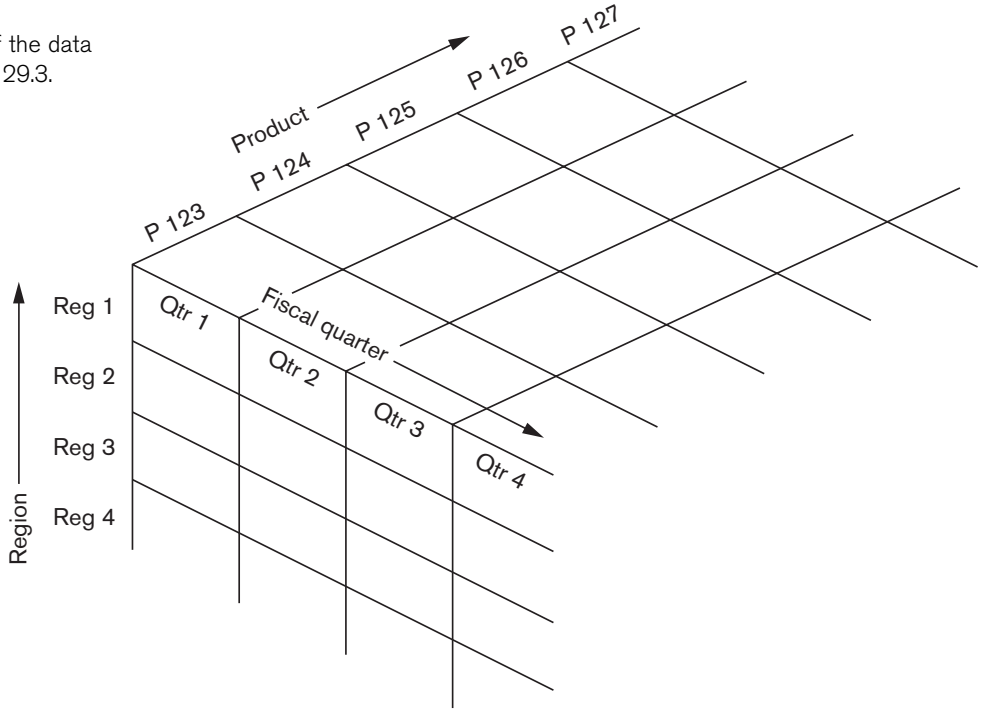


Figure 29.4
Pivoted version of the data cube from Figure 29.3.

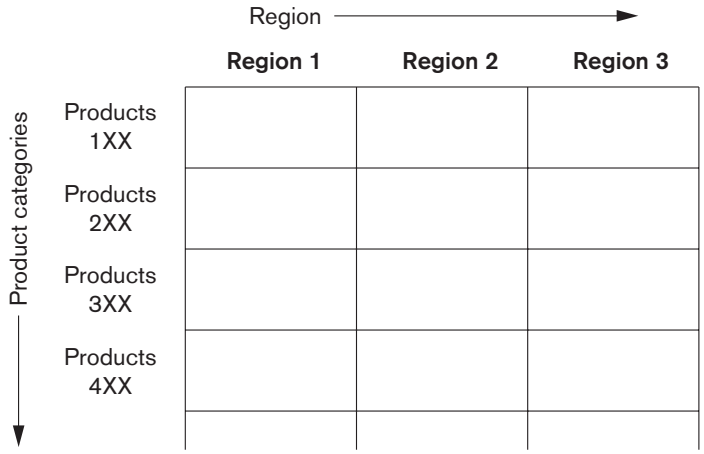


Figure 29.5
The roll-up operation.

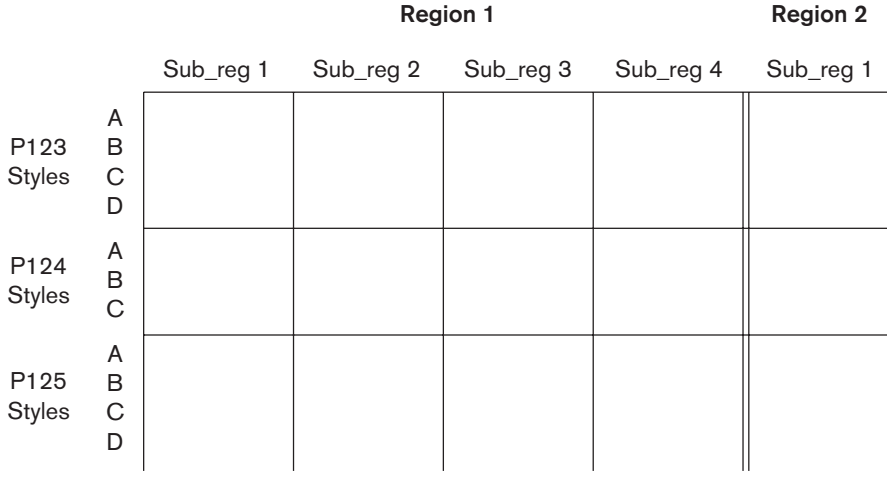


Figure 29.6
The drill-down operation.

The multidimensional storage model involves two types of tables: dimension tables and fact tables. A **dimension table** consists of tuples of attributes of the dimension. A **fact table** can be thought of as having tuples, one per a recorded fact. This fact contains some measured or observed variable(s) and identifies it (them) with pointers to dimension tables. The fact table contains the data, and the dimensions identify each tuple in that data. Figure 29.7 contains an example of a fact table that can be viewed from the perspective of multiple dimension tables.

Two common multidimensional schemas are the star schema and the snowflake schema. The **star schema** consists of a fact table with a single table for each dimension (Figure 29.7). The **snowflake schema** is a variation on the star schema in which

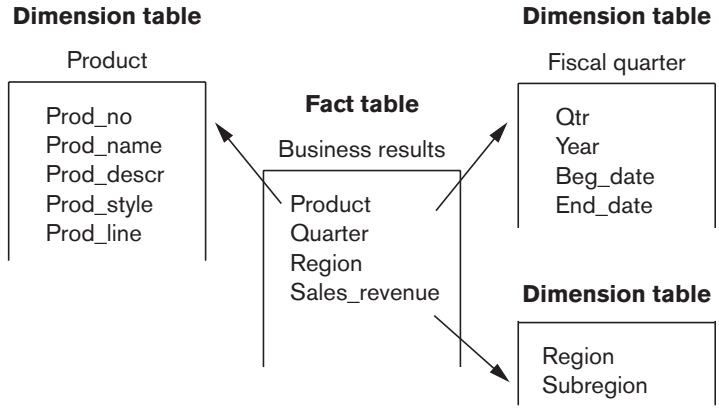
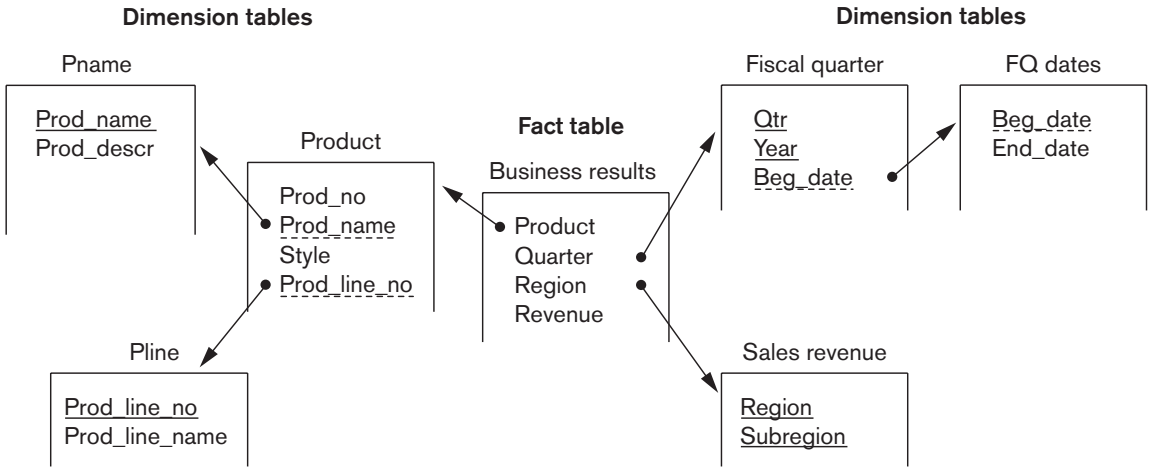


Figure 29.7
A star schema with fact and dimensional tables.

the dimensional tables from a star schema are organized into a hierarchy by normalizing them (Figure 29.8). Some installations are normalizing data warehouses up to the third normal form so that they can access the data warehouse to the finest level of detail. A **fact constellation** is a set of fact tables that share some dimension tables. Figure 29.9 shows a fact constellation with two fact tables, business results and business forecast. These share the dimension table called product. Fact constellations limit the possible queries for the warehouse.

Data warehouse storage also utilizes indexing techniques to support high-performance access (see Chapter 18 for a discussion of indexing). A technique called **bitmap indexing** constructs a bit vector for each value in a domain (column)

Figure 29.8
A snowflake schema.



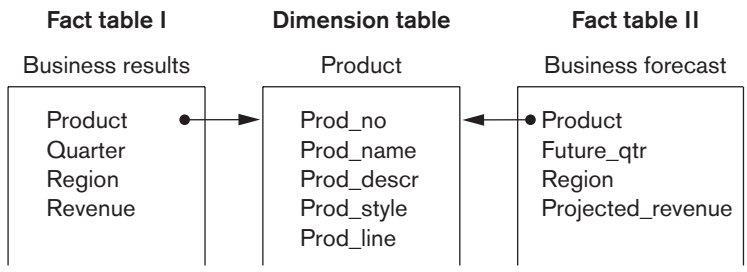


Figure 29.9
A fact constellation.

being indexed. It works very well for domains of low cardinality. There is a 1 bit placed in the *j*th position in the vector if the *j*th row contains the value being indexed. For example, imagine an inventory of 100,000 cars with a bitmap index on car size. If there are four car sizes—economy, compact, mid-size, and full-size—there will be four bit vectors, each containing 100,000 bits (12.5K) for a total index size of 50K. Bitmap indexing can provide considerable input/output and storage space advantages in low-cardinality domains. With bit vectors a bitmap index can provide dramatic improvements in comparison, aggregation, and join performance.

In a star schema, dimensional data can be indexed to tuples in the fact table by **join indexing**. Join indexes are traditional indexes to maintain relationships between primary key and foreign key values. They relate the values of a dimension of a star schema to rows in the fact table. For example, consider a sales fact table that has city and fiscal quarter as dimensions. If there is a join index on city, for each city the join index maintains the tuple IDs of tuples containing that city. Join indexes may involve multiple dimensions.

Data warehouse storage can facilitate access to summary data by taking further advantage of the nonvolatility of data warehouses and a degree of predictability of the analyses that will be performed using them. Two approaches have been used: (1) smaller tables including summary data such as quarterly sales or revenue by product line, and (2) encoding of level (for example, weekly, quarterly, annual) into existing tables. By comparison, the overhead of creating and maintaining such aggregations would likely be excessive in a volatile, transaction-oriented database.

29.4 Building a Data Warehouse

In constructing a data warehouse, builders should take a broad view of the anticipated use of the warehouse. There is no way to anticipate all possible queries or analyses during the design phase. However, the design should specifically support **ad-hoc querying**, that is, accessing data with any meaningful combination of values for the attributes in the dimension or fact tables. For example, a marketing-intensive consumer-products company would require different ways of organizing the data warehouse than would a nonprofit charity focused on fund raising. An appropriate schema should be chosen that reflects anticipated usage.

Acquisition of data for the warehouse involves the following steps:

1. The data must be extracted from multiple, heterogeneous sources, for example, databases or other data feeds such as those containing financial market data or environmental data.
2. Data must be formatted for consistency within the warehouse. Names, meanings, and domains of data from unrelated sources must be reconciled. For instance, subsidiary companies of a large corporation may have different fiscal calendars with quarters ending on different dates, making it difficult to aggregate financial data by quarter. Various credit cards may report their transactions differently, making it difficult to compute all credit sales. These format inconsistencies must be resolved.
3. The data must be cleaned to ensure validity. Data cleaning is an involved and complex process that has been identified as the largest labor-demanding component of data warehouse construction. For input data, cleaning must occur before the data is loaded into the warehouse. There is nothing about cleaning data that is specific to data warehousing and that could not be applied to a host database. However, since input data must be examined and formatted consistently, data warehouse builders should take this opportunity to check for validity and quality. Recognizing erroneous and incomplete data is difficult to automate, and cleaning that requires automatic error correction can be even tougher. Some aspects, such as domain checking, are easily coded into data cleaning routines, but automatic recognition of other data problems can be more challenging. (For example, one might require that City = 'San Francisco' together with State = 'CT' be recognized as an incorrect combination.) After such problems have been taken care of, similar data from different sources must be coordinated for loading into the warehouse. As data managers in the organization discover that their data is being cleaned for input into the warehouse, they will likely want to upgrade their data with the cleaned data. The process of returning cleaned data to the source is called **backflushing** (see Figure 29.1).
4. The data must be fitted into the data model of the warehouse. Data from the various sources must be installed in the data model of the warehouse. Data may have to be converted from relational, object-oriented, or legacy databases (network and/or hierarchical) to a multidimensional model.
5. The data must be loaded into the warehouse. The sheer volume of data in the warehouse makes loading the data a significant task. Monitoring tools for loads as well as methods to recover from incomplete or incorrect loads are required. With the huge volume of data in the warehouse, incremental updating is usually the only feasible approach. The refresh policy will probably emerge as a compromise that takes into account the answers to the following questions:
   - How up-to-date must the data be?
   - Can the warehouse go offline, and for how long?
   - What are the data interdependencies?

- What is the storage availability?
- What are the distribution requirements (such as for replication and partitioning)?
- What is the loading time (including cleaning, formatting, copying, transmitting, and overhead such as index rebuilding)?

As we have said, databases must strike a balance between efficiency in transaction processing and supporting query requirements (ad hoc user requests), but a data warehouse is typically optimized for access from a decision maker's needs. Data storage in a data warehouse reflects this specialization and involves the following processes:

- Storing the data according to the data model of the warehouse
- Creating and maintaining required data structures
- Creating and maintaining appropriate access paths
- Providing for time-variant data as new data are added
- Supporting the updating of warehouse data
- Refreshing the data
- Purging data

Although adequate time can be devoted initially to constructing the warehouse, the sheer volume of data in the warehouse generally makes it impossible to simply reload the warehouse in its entirety later on. Alternatives include selective (partial) refreshing of data and separate warehouse versions (requiring double storage capacity for the warehouse!). When the warehouse uses an incremental data refreshing mechanism, data may need to be periodically purged; for example, a warehouse that maintains data on the previous twelve business quarters may periodically purge its data each year.

Data warehouses must also be designed with full consideration of the environment in which they will reside. Important design considerations include the following:

- Usage projections
- The fit of the data model
- Characteristics of available sources
- Design of the metadata component
- Modular component design
- Design for manageability and change
- Considerations of distributed and parallel architecture

We discuss each of these in turn. Warehouse design is initially driven by usage projections; that is, by expectations about who will use the warehouse and how they will use it. Choice of a data model to support this usage is a key initial decision. Usage projections and the characteristics of the warehouse's data sources are both taken into account. Modular design is a practical necessity to allow the warehouse to evolve with the organization and its information environment. Additionally, a well-

built data warehouse must be designed for maintainability, enabling the warehouse managers to plan for and manage change effectively while providing optimal support to users.

You may recall the term *metadata* from Chapter 1; metadata was defined as the description of a database including its schema definition. The **metadata repository** is a key data warehouse component. The metadata repository includes both technical and business metadata. The first, **technical metadata**, covers details of acquisition processing, storage structures, data descriptions, warehouse operations and maintenance, and access support functionality. The second, **business metadata**, includes the relevant business rules and organizational details supporting the warehouse.

The architecture of the organization's distributed computing environment is a major determining characteristic for the design of the warehouse.

There are two basic distributed architectures: the distributed warehouse and the federated warehouse. For a **distributed warehouse**, all the issues of distributed databases are relevant, for example, replication, partitioning, communications, and consistency concerns. A distributed architecture can provide benefits particularly important to warehouse performance, such as improved load balancing, scalability of performance, and higher availability. A single replicated metadata repository would reside at each distribution site. The idea of the **federated warehouse** is like that of the federated database: a decentralized confederation of autonomous data warehouses, each with its own metadata repository. Given the magnitude of the challenge inherent to data warehouses, it is likely that such federations will consist of smaller scale components, such as data marts. Large organizations may choose to federate data marts rather than build huge data warehouses.

29.5 Typical Functionality of a Data Warehouse

Data warehouses exist to facilitate complex, data-intensive, and frequent ad hoc queries. Accordingly, data warehouses must provide far greater and more efficient query support than is demanded of transactional databases. The data warehouse access component supports enhanced spreadsheet functionality, efficient query processing, structured queries, ad hoc queries, data mining, and materialized views. In particular, enhanced spreadsheet functionality includes support for state-of-the-art spreadsheet applications (for example, MS Excel) as well as for OLAP applications programs. These offer preprogrammed functionalities such as the following:

- **Roll-up.** Data is summarized with increasing generalization (for example, weekly to quarterly to annually).
- **Drill-down.** Increasing levels of detail are revealed (the complement of roll-up).
- **Pivot.** Cross tabulation (also referred to as *rotation*) is performed.
- **Slice and dice.** Projection operations are performed on the dimensions.
- **Sorting.** Data is sorted by ordinal value.

- **Selection.** Data is available by value or range.
- **Derived (computed) attributes.** Attributes are computed by operations on stored and derived values.

Because data warehouses are free from the restrictions of the transactional environment, there is an increased efficiency in query processing. Among the tools and techniques used are query transformation; index intersection and union; special **ROLAP** (relational OLAP) and **MOLAP** (multidimensional OLAP) functions; SQL extensions; advanced join methods; and intelligent scanning (as in piggy-backing multiple queries).

Improved performance has also been attained with parallel processing. Parallel server architectures include symmetric multiprocessor (SMP), cluster, and massively parallel processing (MPP), and combinations of these.

Knowledge workers and decision makers use tools ranging from parametric queries to ad hoc queries to data mining. Thus, the access component of the data warehouse must provide support for structured queries (both parametric and ad hoc). Together, these make up a managed query environment. Data mining itself uses techniques from statistical analysis and artificial intelligence. Statistical analysis can be performed by advanced spreadsheets, by sophisticated statistical analysis software, or by custom-written programs. Techniques such as lagging, moving averages, and regression analysis are also commonly employed. Artificial intelligence techniques, which may include genetic algorithms and neural networks, are used for classification and are employed to discover knowledge from the data warehouse that may be unexpected or difficult to specify in queries. (We treat data mining in detail in Chapter 28.)

29.6 Data Warehouse versus Views

Some people have considered data warehouses to be an extension of database views. Earlier we mentioned materialized views as one way of meeting requirements for improved access to data (see Section 5.3 for a discussion of views). Materialized views have been explored for their performance enhancement. Views, however, provide only a subset of the functions and capabilities of data warehouses. Views and data warehouses are alike in that they both have read-only extracts from databases and subject orientation. However, data warehouses are different from views in the following ways:

- Data warehouses exist as persistent storage instead of being materialized on demand.
- Data warehouses are not usually relational, but rather multidimensional. Views of a relational database are relational.
- Data warehouses can be indexed to optimize performance. Views cannot be indexed independent of the underlying databases.
- Data warehouses characteristically provide specific support of functionality; views cannot.

- Data warehouses provide large amounts of integrated and often temporal data, generally more than is contained in one database, whereas views are an extract of a database.

29.7 Difficulties of Implementing Data Warehouses

Some significant operational issues arise with data warehousing: construction, administration, and quality control. Project management—the design, construction, and implementation of the warehouse—is an important and challenging consideration that should not be underestimated. The building of an enterprise-wide warehouse in a large organization is a major undertaking, potentially taking years from conceptualization to implementation. Because of the difficulty and amount of lead time required for such an undertaking, the widespread development and deployment of data marts may provide an attractive alternative, especially to those organizations with urgent needs for OLAP, DSS, and/or data mining support.

The administration of a data warehouse is an intensive enterprise, proportional to the size and complexity of the warehouse. An organization that attempts to administer a data warehouse must realistically understand the complex nature of its administration. Although designed for read access, a data warehouse is no more a static structure than any of its information sources. Source databases can be expected to evolve. The warehouse's schema and acquisition component must be expected to be updated to handle these evolutions.

A significant issue in data warehousing is the quality control of data. Both quality and consistency of data are major concerns. Although the data passes through a cleaning function during acquisition, quality and consistency remain significant issues for the database administrator. Melding data from heterogeneous and disparate sources is a major challenge given differences in naming, domain definitions, identification numbers, and the like. Every time a source database changes, the data warehouse administrator must consider the possible interactions with other elements of the warehouse.

Usage projections should be estimated conservatively prior to construction of the data warehouse and should be revised continually to reflect current requirements. As utilization patterns become clear and change over time, storage and access paths can be tuned to remain optimized for support of the organization's use of its warehouse. This activity should continue throughout the life of the warehouse in order to remain ahead of demand. The warehouse should also be designed to accommodate the addition and attrition of data sources without major redesign. Sources and source data will evolve, and the warehouse must accommodate such change. Fitting the available source data into the data model of the warehouse will be a continual challenge, a task that is as much art as science. Because there is continual rapid change in technologies, both the requirements and capabilities of the warehouse will change considerably over time. Additionally, data warehousing technology itself will continue to evolve for some time so that component structures and functional-

ities will continually be upgraded. This certain change is excellent motivation for having fully modular design of components.

Administration of a data warehouse will require far broader skills than are needed for traditional database administration. A team of highly skilled technical experts with overlapping areas of expertise will likely be needed, rather than a single individual. Like database administration, data warehouse administration is only partly technical; a large part of the responsibility requires working effectively with all the members of the organization with an interest in the data warehouse. However difficult that can be at times for database administrators, it is that much more challenging for data warehouse administrators, as the scope of their responsibilities is considerably broader.

Design of the management function and selection of the management team for a database warehouse are crucial. Managing the data warehouse in a large organization will surely be a major task. Many commercial tools are available to support management functions. Effective data warehouse management will certainly be a team function, requiring a wide set of technical skills, careful coordination, and effective leadership. Just as we must prepare for the evolution of the warehouse, we must also recognize that the skills of the management team will, of necessity, evolve with it.

29.8 Summary

In this chapter we surveyed the field known as data warehousing. Data warehousing can be seen as a process that requires a variety of activities to precede it. In contrast, data mining (see Chapter 28) may be thought of as an activity that draws knowledge from an existing data warehouse. We introduced key concepts related to data warehousing and we discussed the special functionality associated with a multidimensional view of data. We also discussed the ways in which data warehouses supply decision makers with information at the correct level of detail, based on an appropriate organization and perspective.

Review Questions

29.1. What is a data warehouse? How does it differ from a database?

29.2. Define the terms: OLAP (online analytical processing), ROLAP (relational OLAP), MOLAP (multidimensional OLAP), and DSS (decision-support systems).

29.3. Describe the characteristics of a data warehouse. Divide them into functionality of a warehouse and advantages users derive from it.

29.4. What is the multidimensional data model? How is it used in data warehousing?

29.5. Define the following terms: star schema, snowflake schema, fact constellation, data marts.

29.6. What types of indexes are built for a warehouse? Illustrate the uses for each with an example.

29.7. Describe the steps of building a warehouse.

29.8. What considerations play a major role in the design of a warehouse?

29.9. Describe the functions a user can perform on a data warehouse and illustrate the results of these functions on a sample multidimensional data warehouse.

29.10. How is the concept of a relational view related to a data warehouse and data marts? In what way are they different?

29.11. List the difficulties in implementing a data warehouse.

29.12. List the open issues and research problems in data warehousing.

Selected Bibliography

Inmon (1992, 2005) is credited for giving the term wide acceptance. Codd and Salley (1993) popularized the term online analytical processing (OLAP) and defined a set of characteristics for data warehouses to support OLAP. Kimball (1996) is known for his contribution to the development of the data warehousing field. Mattison (1996) is one of the several books on data warehousing that gives a comprehensive analysis of techniques available in data warehouses and the strategies companies should use in deploying them. Ponniah (2002) gives a very good practical overview of the data warehouse building process from requirements collection to deployment maintenance. Bischoff and Alexander (1997) is a compilation of advice from experts. Chaudhuri and Dayal (1997) give an excellent tutorial on the topic, while Widom (1995) points to a number of outstanding research problems.

appendix A

Alternative Diagrammatic Notations for ER Models

Figure A.1 shows a number of different diagrammatic notations for representing ER and EER model concepts. Unfortunately, there is no standard notation: different database design practitioners prefer different notations. Similarly, various **CASE** (computer-aided software engineering) tools and **OOA** (object-oriented analysis) methodologies use various notations. Some notations are associated with models that have additional concepts and constraints beyond those of the ER and EER models described in Chapters 7 through 9, while other models have fewer concepts and constraints. The notation we used in Chapter 7 is quite close to the original notation for ER diagrams, which is still widely used. We discuss some alternate notations here.

Figure A.1(a) shows different notations for displaying entity types/classes, attributes, and relationships. In Chapters 7 through 9, we used the symbols marked (i) in Figure A.1(a)—namely, rectangle, oval, and diamond. Notice that symbol (ii) for entity types/classes, symbol (ii) for attributes, and symbol (ii) for relationships are similar, but they are used by different methodologies to represent three different concepts. The straight line symbol (iii) for representing relationships is used by several tools and methodologies.

Figure A.1(b) shows some notations for attaching attributes to entity types. We used notation (i). Notation (ii) uses the third notation (iii) for attributes from Figure A.1(a). The last two notations in Figure A.1(b)—(iii) and (iv)—are popular in OOA methodologies and in some CASE tools. In particular, the last notation displays both the attributes and the methods of a class, separated by a horizontal line.

Figure A.1
Alternative notations. (a) Symbols for entity type/class, attribute, and relationship. (b) Displaying attributes. (c) Displaying cardinality ratios. (d) Various (min, max) notations. (e) Notations for displaying specialization/generalization.

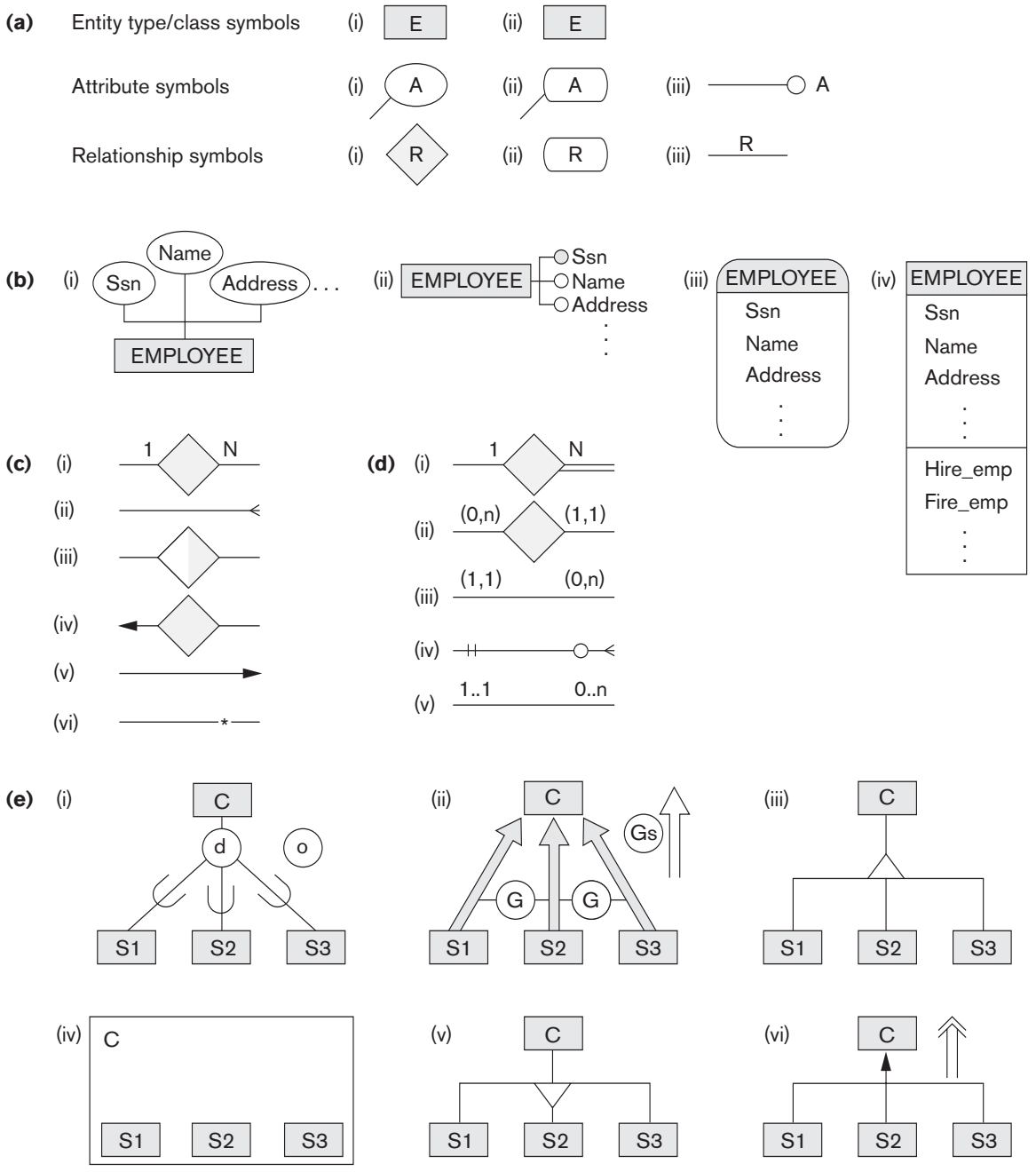


Figure A.1(c) shows various notations for representing the cardinality ratio of binary relationships. We used notation (i) in Chapters 7 through 9. Notation (ii)—known as the *chicken feet* notation—is quite popular. Notation (iv) uses the arrow as a functional reference (from the N to the 1 side) and resembles our notation for foreign keys in the relational model (see Figure 9.2); notation (v)—used in *Bachman diagrams* and the network data model—uses the arrow in the *reverse direction* (from the 1 to the N side). For a 1:1 relationship, (ii) uses a straight line without any chicken feet; (iii) makes both halves of the diamond white; and (iv) places arrowheads on both sides. For an M:N relationship, (ii) uses chicken feet at both ends of the line; (iii) makes both halves of the diamond black; and (iv) does not display any arrowheads.

Figure A.1(d) shows several variations for displaying (min, max) constraints, which are used to display both cardinality ratio and total/partial participation. We mostly used notation (i). Notation (ii) is the alternative notation we used in Figure 7.15 and discussed in Section 7.7.4. Recall that our notation specifies the constraint that each entity must participate in at least min and at most max relationship instances. Hence, for a 1:1 relationship, both max values are 1; for M:N, both max values are n. A min value greater than 0 (zero) specifies total participation (existence dependency). In methodologies that use the straight line for displaying relationships, it is common to *reverse the positioning* of the (min, max) constraints, as shown in (iii); a variation common in some tools (and in UML notation) is shown in (v). Another popular technique—which follows the same positioning as (iii)—is to display the *min* as o ("oh" or circle, which stands for zero) or as | (vertical dash, which stands for 1), and to display the max as | (vertical dash, which stands for 1) or as chicken feet (which stands for n), as shown in (iv).

Figure A.1(e) shows some notations for displaying specialization/generalization. We used notation (i) in Chapter 8, where a d in the circle specifies that the subclasses (S1, S2, and S3) are disjoint and an o in the circle specifies overlapping subclasses. Notation (ii) uses G (for generalization) to specify disjoint, and Gs to specify overlapping; some notations use the solid arrow, while others use the empty arrow (shown at the side). Notation (iii) uses a triangle pointing toward the superclass, and notation (v) uses a triangle pointing toward the subclasses; it is also possible to use both notations in the same methodology, with (iii) indicating generalization and (v) indicating specialization. Notation (iv) places the boxes representing subclasses within the box representing the superclass. Of the notations based on (vi), some use a single-lined arrow, and others use a double-lined arrow (shown at the side).

The notations shown in Figure A.1 show only some of the diagrammatic symbols that have been used or suggested for displaying database conceptual schemes. Other notations, as well as various combinations of the preceding, have also been used. It would be useful to establish a standard that everyone would adhere to, in order to prevent misunderstandings and reduce confusion.

appendix B

Parameters of Disks

The most important disk parameter is the time required to locate an arbitrary disk block, given its block address, and then to transfer the block between the disk and a main memory buffer. This is the random access time for accessing a disk block. There are three time components to consider as follows:

1. **Seek time (*s*).** This is the time needed to mechanically position the read/write head on the correct track for movable-head disks. (For fixed-head disks, it is the time needed to electronically switch to the appropriate read/write head.) For movable-head disks, this time varies, depending on the distance between the current track under the read/write head and the track specified in the block address. Usually, the disk manufacturer provides an average seek time in milliseconds. The typical range of average seek time is 4 to 10 msec. This is the main *culprit* for the delay involved in transferring blocks between disk and memory.
2. **Rotational delay (*rd*).** Once the read/write head is at the correct track, the user must wait for the beginning of the required block to rotate into position under the read/write head. On average, this takes about the time for half a revolution of the disk, but it actually ranges from immediate access (if the start of the required block is in position under the read/write head right after the seek) to a full disk revolution (if the start of the required block just passed the read/write head after the seek). If the speed of disk rotation is p revolutions per minute (rpm), then the average rotational delay rd is given by

   $$rd = (1/2) * (1/p) \text{ min} = (60 * 1000)/(2 * p) \text{ msec} = 30000/p \text{ msec}$$

   A typical value for p is 10,000 rpm, which gives a rotational delay of $rd = 3$ msec. For fixed-head disks, where the seek time is negligible, this component causes the greatest delay in transferring a disk block.

3. **Block transfer time (*btt*).** Once the read/write head is at the beginning of the required block, some time is needed to transfer the data in the block. This block transfer time depends on the block size, track size, and rotational speed. If the **transfer rate** for the disk is *tr* bytes/msec and the block size is *B* bytes, then

   $btt = B/tr$ msec

   If we have a track size of 50 Kbytes and *p* is 3600 rpm, then the transfer rate in bytes/msec is

   $tr = (50 * 1000)/(60 * 1000/3600) = 3000$ bytes/msec

   In this case, $btt = B/3000$ msec, where *B* is the block size in bytes.

The average time (*s*) needed to find and transfer a block, given its block address, is estimated by

$(s + rd + btt)$ msec

This holds for either reading or writing a block. The principal method of reducing this time is to transfer several blocks that are stored on one or more tracks of the same cylinder; then the seek time is required for the first block only. To transfer consecutively *k noncontiguous* blocks that are on the same cylinder, we need approximately

$s + (k * (rd + btt))$ msec

In this case, we need two or more buffers in main storage because we are continuously reading or writing the *k* blocks, as we discussed in Chapter 17. The transfer time per block is reduced even further when *consecutive blocks* on the same track or cylinder are transferred. This eliminates the rotational delay for all but the first block, so the estimate for transferring *k* consecutive blocks is

$s + rd + (k * btt)$ msec

A more accurate estimate for transferring consecutive blocks takes into account the interblock gap (see Section 17.2.1), which includes the information that enables the read/write head to determine which block it is about to read. Usually, the disk manufacturer provides a **bulk transfer rate (*btr*)** that takes the gap size into account when reading consecutively stored blocks. If the gap size is *G* bytes, then

$btr = (B/(B + G)) * tr$ bytes/msec

The bulk transfer rate is the rate of transferring *useful bytes* in the data blocks. The disk read/write head must go over all bytes on a track as the disk rotates, including the bytes in the interblock gaps, which store control information but not real data. When the bulk transfer rate is used, the time needed to transfer the useful data in one block out of several consecutive blocks is B/btr. Hence, the estimated time to read *k* blocks consecutively stored on the same cylinder becomes

$s + rd + (k * (B/btr))$ msec

Another parameter of disks is the **rewrite time**. This is useful in cases when we read a block from the disk into a main memory buffer, update the buffer, and then write the buffer back to the same disk block on which it was stored. In many cases, the time required to update the buffer in main memory is less than the time required for one disk revolution. If we know that the buffer is ready for rewriting, the system can keep the disk heads on the same track, and during the next disk revolution the updated buffer is rewritten back to the disk block. Hence, the rewrite time T_{rw}, is usually estimated to be the time needed for one disk revolution:

$$T_{rw} = 2 * rd \text{ msec} = 60000/p \text{ msec}$$

To summarize, the following is a list of the parameters we have discussed and the symbols we use for them:

Seek time:	s msec
Rotational delay:	rd msec
Block transfer time:	btt msec
Rewrite time:	T_{rw} msec
Transfer rate:	tr bytes/msec
Bulk transfer rate:	btr bytes/msec
Block size:	B bytes
Interblock gap size:	G bytes
Disk speed:	p rpm (revolutions per minute)

appendix C

Overview of the QBE Language

The Query-By-Example (QBE) language is important because it is one of the first graphical query languages with minimum syntax developed for database systems. It was developed at IBM Research and is available as an IBM commercial product as part of the QMF (Query Management Facility) interface option to DB2. The language was also implemented in the Paradox DBMS, and is related to a point-and-click type interface in the Microsoft Access DBMS. It differs from SQL in that the user does not have to explicitly specify a query using a fixed syntax; rather, the query is formulated by filling in **templates** of relations that are displayed on a monitor screen. Figure C.1 shows how these templates may look for the database of Figure 3.5. The user does not have to remember the names of attributes or relations because they are displayed as part of these templates. Additionally, the user does not have to follow rigid syntax rules for query specification; rather, constants and variables are entered in the columns of the templates to construct an **example** related to the retrieval or update request. QBE is related to the domain relational calculus, as we shall see, and its original specification has been shown to be relationally complete.

C.1 Basic Retrievals in QBE

In QBE retrieval queries are specified by filling in one or more rows in the templates of the tables. For a single relation query, we enter either constants or **example elements** (a QBE term) in the columns of the template of that relation. An example element stands for a domain variable and is specified as an example value preceded by the underscore character (_). Additionally, a P. prefix (called the P dot operator) is entered in certain columns to indicate that we would like to print (or display)

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date

DEPT_LOCATIONS

Dnumber	Dlocation

PROJECT

Pname	Pnumber	Plocation	Dnum

WORKS_ON

Essn	Pno	Hours

DEPENDENT

Essn	Dependent_name	Sex	Bdate	Relationship

Figure C.1
The relational schema of Figure 3.5 as it may be displayed by QBE.

values in those columns for our result. The constants specify values that must be exactly matched in those columns.

For example, consider the query Q0: *Retrieve the birth date and address of John B. Smith.* In Figures C.2(a) through C.2(d) we show how this query can be specified in a progressively more terse form in QBE. In Figure C.2(a) an example of an employee is presented as the type of row that we are interested in. By leaving John B. Smith as constants in the Fname, Minit, and Lname columns, we are specifying an exact match in those columns. The rest of the columns are preceded by an underscore indicating that they are domain variables (example elements). The P. prefix is placed in the Bdate and Address columns to indicate that we would like to output value(s) in those columns.

Q0 can be abbreviated as shown in Figure C.2(b). There is no need to specify example values for columns in which we are not interested. Moreover, because example values are completely arbitrary, we can just specify variable names for them, as shown in Figure C.2(c). Finally, we can also leave out the example values entirely, as shown in Figure C.2(d), and just specify a P. under the columns to be retrieved.

To see how retrieval queries in QBE are similar to the domain relational calculus, compare Figure C.2(d) with Q0 (simplified) in domain calculus as follows:

Q0 : { uv | EMPLOYEE(qrstuvwxyz) **and** *q*='John' **and** *r*='B' **and** *s*='Smith'}

(a) **EMPLOYEE**

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	_123456789	P._9/1/60	P._100 Main, Houston, TX	_M	_25000	_123456789	_3

(b) **EMPLOYEE**

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith		P._9/1/60	P._100 Main, Houston, TX				

(c) **EMPLOYEE**

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith		P._X	P._Y				

(d) **EMPLOYEE**

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith		P.	P.				

Figure C.2
Four ways to specify the query Q0 in QBE.

We can think of each column in a QBE template as an *implicit domain variable;* hence, Fname corresponds to the domain variable *q*, Minit corresponds to *r*, ..., and Dno corresponds to *z*. In the QBE query, the columns with P. correspond to variables specified to the left of the bar in domain calculus, whereas the columns with constant values correspond to tuple variables with equality selection conditions on them. The condition EMPLOYEE(qrstuvwxyz) and the existential quantifiers are implicit in the QBE query because the template corresponding to the EMPLOYEE relation is used.

In QBE, the user interface first allows the user to choose the tables (relations) needed to formulate a query by displaying a list of all relation names. Then the templates for the chosen relations are displayed. The user moves to the appropriate columns in the templates and specifies the query. Special function keys are provided to move among templates and perform certain functions.

We now give examples to illustrate basic facilities of QBE. Comparison operators other than = (such as > or ≥) may be entered in a column before typing a constant value. For example, the query Q0A: *List the social security numbers of employees who work more than 20 hours per week on project number 1* can be specified as shown in Figure C.3(a). For more complex conditions, the user can ask for a **condition box**, which is created by pressing a particular function key. The user can then type the complex condition.[1]

[1]Negation with the ¬ symbol is not allowed in a condition box.

Figure C.3
Specifying complex conditions in QBE. (a) The query Q0A. (b) The query Q0B with a condition box. (c) The query Q0B without a condition box.

(a)

WORKS_ON

Essn	Pno	Hours
P.		> 20

(b)

WORKS_ON

Essn	Pno	Hours
P.	_PX	_HX

CONDITIONS

_HX > 20 and (PX = 1 or PX = 2)

(c)

WORKS_ON

Essn	Pno	Hours
P.	1	> 20
P.	2	> 20

For example, the query Q0B: *List the social security numbers of employees who work more than 20 hours per week on either project 1 or project 2* can be specified as shown in Figure C.3(b).

Some complex conditions can be specified without a condition box. The rule is that all conditions specified on the same row of a relation template are connected by the **and** logical connective (*all* must be satisfied by a selected tuple), whereas conditions specified on distinct rows are connected by **or** (*at least one* must be satisfied). Hence, Q0B can also be specified, as shown in Figure C.3(c), by entering two distinct rows in the template.

Now consider query Q0C: *List the social security numbers of employees who work on both project 1 and project 2*; this cannot be specified as in Figure C.4(a), which lists those who work on *either* project 1 or project 2. The example variable _ES will bind itself to Essn values in <–, 1, –> tuples *as well as* to those in <–, 2, –> tuples. Figure C.4(b) shows how to specify Q0C correctly, where the condition (_EX = _EY) in the box makes the _EX and _EY variables bind only to identical Essn values.

In general, once a query is specified, the resulting values are displayed in the template under the appropriate columns. If the result contains more rows than can be displayed on the screen, most QBE implementations have function keys to allow scrolling up and down the rows. Similarly, if a template or several templates are too wide to appear on the screen, it is possible to scroll sideways to examine all the templates.

A join operation is specified in QBE by using the *same variable*[2] in the columns to be joined. For example, the query Q1: *List the name and address of all employees who*

[2]A variable is called an **example element** in QBE manuals.

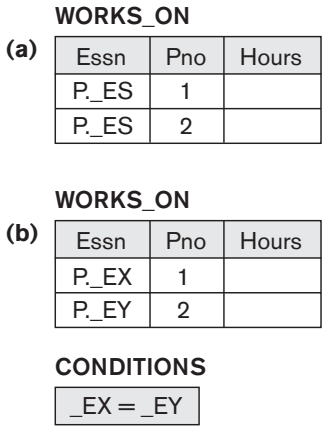


Figure C.4
Specifying EMPLOYEES who work on both projects. (a) Incorrect specification of an AND condition. (b) Correct specification.

work for the 'Research' department can be specified as shown in Figure C.5(a). Any number of joins can be specified in a single query. We can also specify a **result table** to display the result of the join query, as shown in Figure C.5(a); this is needed if the result includes attributes from two or more relations. If no result table is specified, the system provides the query result in the columns of the various relations, which may make it difficult to interpret. Figure C.5(a) also illustrates the feature of QBE for specifying that all attributes of a relation should be retrieved, by placing the P. operator under the relation name in the relation template.

To join a table with itself, we specify different variables to represent the different references to the table. For example, query Q8: *For each employee retrieve the employee's first and last name as well as the first and last name of his or her immediate supervisor* can be specified as shown in Figure C.5(b), where the variables starting with E refer to an employee and those starting with S refer to a supervisor.

C.2 Grouping, Aggregation, and Database Modification in QBE

Next, consider the types of queries that require grouping or aggregate functions. A grouping operator *G.* can be specified in a column to indicate that tuples should be grouped by the value of that column. Common functions can be specified, such as AVG., SUM., CNT. (count), MAX., and MIN. In QBE the functions AVG., SUM., and CNT. are applied to distinct values within a group in the default case. If we want these functions to apply to all values, we must use the prefix ALL.[3] This convention is *different* in SQL, where the default is to apply a function to all values.

[3]ALL in QBE is unrelated to the universal quantifier.

(a) **EMPLOYEE**

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
_FN		_LN			_Addr				_DX

DEPARTMENT

Dname	Dnumber	Mgrssn	Mgr_start_date
Research	_DX		

RESULT			
P.	_FN	_LN	_Addr

(b) **EMPLOYEE**

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
_E1		_E2						_Xssn	
_S1		_S2	_Xssn						

RESULT				
P.	_E1	_E2	_S1	_S2

Figure C.5
Illustrating JOIN and result relations in QBE. (a) The query Q1. (b) The query Q8.

Figure C.6(a) shows query Q23, which counts the number of *distinct* salary values in the EMPLOYEE relation. Query Q23A (Figure C.6(b) counts all salary values, which is the same as counting the number of employees. Figure C.6(c) shows Q24, which retrieves each department number and the number of employees and average salary within each department; hence, the Dno column is used for grouping as indicated by the *G.* function. Several of the operators G., P., and ALL can be specified in a single column. Figure C.6(d) shows query Q26, which displays each project name and the number of employees working on it for projects on which more than two employees work.

QBE has a negation symbol, ¬, which is used in a manner similar to the NOT EXISTS function in SQL. Figure C.7 shows query Q6, which lists the names of employees who have no dependents. The negation symbol ¬ says that we will select values of the _SX variable from the EMPLOYEE relation only if they do not occur in the DEPENDENT relation. The same effect can be produced by placing a ¬ _SX in the Essn column.

Although the QBE language as originally proposed was shown to support the equivalent of the EXISTS and NOT EXISTS functions of SQL, the QBE implementation in QMF (under the DB2 system) does *not* provide this support. Hence, the QMF version of QBE, which we discuss here, is *not relationally complete.* Queries such as Q3: *Find employees who work on all projects controlled by department 5* cannot be specified.

(a) EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
							P.CNT.		

(b) EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
							P.CNT.ALL		

(c) EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
			P.CNT.ALL				P.AVG.ALL		P.G.

(d) PROJECT

Pname	Pnumber	Plocation	Dnum
P.	_PX		

WORKS_ON

Essn	Pno	Hours
P.CNT.EX	G._PX	

CONDITIONS

CNT._EX > 2

Figure C.6
Functions and grouping in QBE. (a) The query Q23. (b) The query Q23A. (c) The query Q24. (d) The query Q26.

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
P.		P.	_SX						

DEPENDENT

Essn	Dependent_name	Sex	Bdate	Relationship
_SX				

Figure C.7
Illustrating negation by the query Q6.

There are three QBE operators for modifying the database: I. for insert, D. for delete, and U. for update. The insert and delete operators are specified in the template column under the relation name, whereas the update operator is specified under the columns to be updated. Figure C.8(a) shows how to insert a new EMPLOYEE tuple. For deletion, we first enter the D. operator and then specify the tuples to be deleted by a condition (Figure C.8(b)). To update a tuple, we specify the U. operator under the attribute name, followed by the new value of the attribute. We should also select the tuple or tuples to be updated in the usual way. Figure C.8(c) shows an update

(a) **EMPLOYEE**

	Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
I.	Richard	K	Marini	653298653	30-Dec-52	98 Oak Forest, Katy, TX	M	37000	987654321	4

(b) **EMPLOYEE**

	Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
D.				653298653						

(c) **EMPLOYEE**

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John		Smith					U._S*1.1		U.4

Figure C.8
Modifying the database in QBE. (a) Insertion. (b) Deletion. (c) Update in QBE.

request to increase the salary of 'John Smith' by 10 percent and also to reassign him to department number 4.

QBE also has data definition capabilities. The tables of a database can be specified interactively, and a table definition can also be updated by adding, renaming, or removing a column. We can also specify various characteristics for each column, such as whether it is a key of the relation, what its data type is, and whether an index should be created on that field. QBE also has facilities for view definition, authorization, storing query definitions for later use, and so on.

QBE does not use the *linear* style of SQL; rather, it is a *two-dimensional* language because users specify a query moving around the full area of the screen. Tests on users have shown that QBE is easier to learn than SQL, especially for nonspecialists. In this sense, QBE was the *first* user-friendly *visual* relational database language.

More recently, numerous other user-friendly interfaces have been developed for commercial database systems. The use of menus, graphics, and forms is now becoming quite common. Filling forms partially to issue a search request is akin to using QBE. Visual query languages, which are still not so common, are likely to be offered with commercial relational databases in the future.