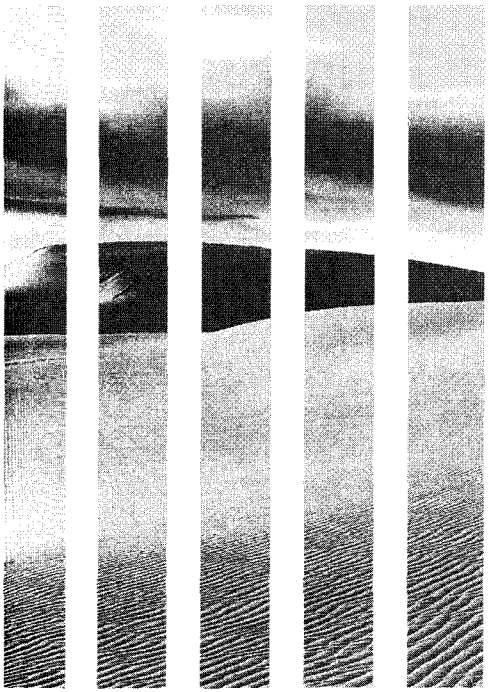
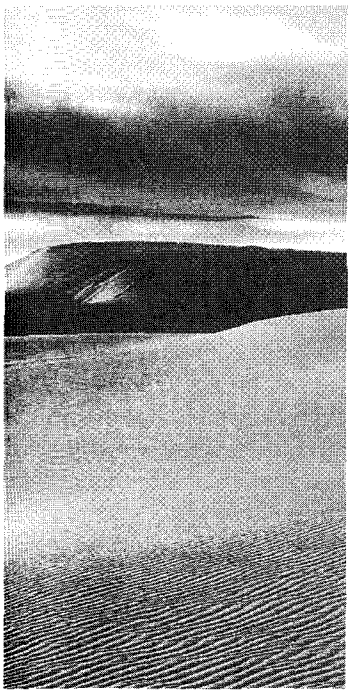


4

DATA STORAGE, INDEXING, QUERY PROCESSING, AND PHYSICAL DESIGN



13

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. We start in Section 13.1 by introducing the concepts of computer storage hierarchies and how they are used in database systems. Section 13.2 is devoted to a description of magnetic disk storage devices and their characteristics, and we also briefly describe magnetic tape storage devices. Having discussed different storage technologies, we then turn our attention to the methods for organizing data on disks. Section 13.3 covers the technique of double buffering, which is used to speed retrieval of multiple disk blocks. In Section 13.4 we discuss various ways of formatting and storing records of a file on disk. Section 13.5 discusses the various types of operations that are typically applied to records of a file. We then present three primary methods for organizing records of a file on disk: unordered records, discussed in Section 13.6; ordered records, in Section 13.7; and hashed records, in Section 13.8.

Section 13.9 very briefly discusses 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 discuss later in Chapters 20 and 21. Section 13.9 describes RAID (Redundant Arrays of Inexpensive (or Independent) Disks)—a data storage system architecture that is used commonly in large organizations for better reliability and performance. Finally, in Section 13.10 we describe storage area networks, a more recent approach for managing stored data on networks. In Chapter 14 we discuss

techniques for creating auxiliary data structures, called indexes, that 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 13 and 14 may be browsed through or even omitted by readers who have already studied file organizations. The material covered here is necessary for understanding Chapters 15 and 16 that deal with query processing and query optimization.

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

We will first give an overview of the various storage devices used for primary and secondary storage in Section 13.1.1 and will then discuss how databases are typically handled in the storage hierarchy in Section 13.1.2.

13.1.1 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 programs. The next level of primary storage is DRAM (Dynamic RAM), which provides the main work area for the CPU for keeping programs and data and 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 storage level*, the hierarchy includes magnetic disks, as well as **mass storage** in the form of CD-ROM (Compact Disk–Read-Only

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

Memory) devices, and finally tapes at the least expensive end of the hierarchy. The **storage capacity** is measured in kilobytes (Kbyte or 1000 bytes), megabytes (Mbyte or 1 million bytes), gigabytes (Gbyte or 1 billion bytes), and even terabytes (1000 Gbytes).

Programs reside and execute in DRAM. Generally, large permanent databases reside on secondary storage, and portions of the database are read into and written from buffers in main memory as needed. Now that personal computers and workstations have hundreds of megabytes of data in DRAM, it is becoming possible to load a large fraction of the database into main memory. Eight to 16 gigabytes of RAM on a single server are 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 at a time.[2] 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, USB storage accessories, etc.

CD-ROM 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 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. **Optical juke box memories** use an array of CD-ROM platters, which are loaded onto drives on demand. Although optical juke boxes have capacities in the hundreds of gigabytes, their retrieval times are in the hundreds of milliseconds, quite a bit slower than magnetic disks.[3] This type of storage is continuing to decline because of the rapid decrease in cost and increase in capacities of magnetic disks. The DVD (Digital Video Disk) is a recent standard for optical disks allowing 4.5 to 15 gigabytes of storage per disk. Most personal computer disk drives now read CD-ROM and DVD disks.

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** cannot be defined precisely any more because

2. For example, the INTEL DD28F032SA is a 32-megabit capacity flash memory with 70-nanosecond access speed, and 430 KB/second write transfer rate.

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

disk storage capacities are on the rise and costs are declining. It may very soon be reserved for databases containing tens of terabytes.

13.1.2 Storage of Databases

Databases typically store large amounts of data that must *persist* over long periods of time. The data is accessed and processed repeatedly during this period. This contrasts with the notion of *transient* data structures 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 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. Databases in the future may therefore reside at different levels of the memory hierarchy from those described in Section 13.1.1. However, it is anticipated that magnetic disks will continue to be the medium of primary 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 the database because storage on tape costs even less than storage on disk. However, access to data on tape is quite slow. Data stored on tapes is **offline;** that is, some intervention by an operator—or an automatic loading device—to load a tape is needed before this 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, and the process of **physical database design** involves choosing from among the options the particular data organization techniques that best suit the given application requirements. DBMS system implementers must study data organization techniques so that they can implement them efficiently and thus provide the DBA and users of the DBMS with sufficient options.

Typical database applications need only a small portion of the database at a time for processing. Whenever a certain portion of the data is needed, it must be located on disk, copied to main memory for processing, and then rewritten to the disk if the data is changed. The data stored on disk is organized as **files** of **records.** Each record is a

collection of data values that can be interpreted as facts about entities, their attributes, and their relationships. Records should be stored on disk in a manner that makes it possible to locate them efficiently whenever they are needed.

There are several **primary file organizations,** which determine how the records of a file are *physically placed* on the disk, *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 13.6 through 13.9. A **secondary organization** or **auxiliary access structure** allows efficient access to the records of a file 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 14.

13.2 Secondary Storage Devices

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

13.2.1 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 Kbytes to 1.5 Mbytes; hard disks for micros typically hold from several hundred Mbytes up to a few Gbytes; and large disk packs used with servers and mainframes have capacities that range up to a few tens or hundreds of Gbytes. Disk capacities continue to grow as technology improves.

Whatever their capacity, disks are all made of magnetic material shaped as a thin circular disk (Figure 13.1a) and protected by a plastic or acrylic cover. A disk is **single-sided** if it stores information on only one of its surfaces and **double-sided** if both surfaces are used. To increase storage capacity, disks are assembled into a **disk pack** (Figure 13.1b), which may include many disks and hence many surfaces. Information is stored on a disk surface in concentric circles of *small width*,[4] each having a distinct diameter. Each circle is

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

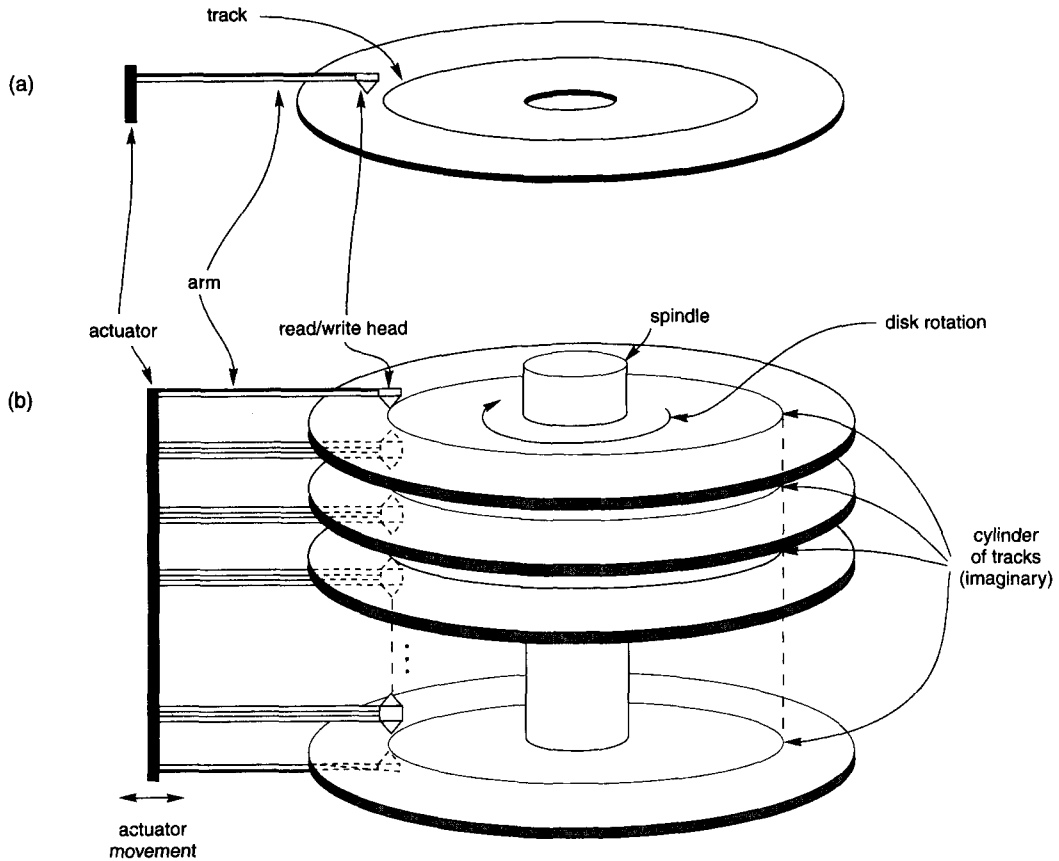


FIGURE 13.1 (a) A single-sided disk with read/write hardware. (b) A disk pack with read/write hardware.

called a **track.** For disk packs, the 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 calls a portion of a track that subtends a fixed angle at the center as a sector (Figure 13.2a). 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 (Figure 13.2b). A technique called ZBR (Zone Bit Recording) allows a range of cylinders to have the same number of

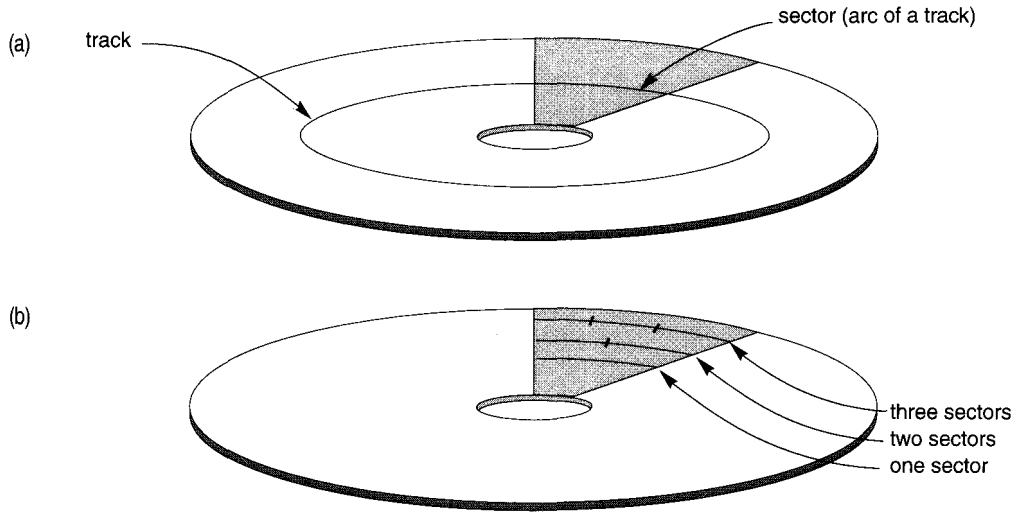


FIGURE 13.2 Different sector organizations on disk. (a) Sectors subtending a fixed angle. (b) Sectors maintaining a uniform recording density.

sectors per arc. For example, cylinders 0–99 may have one sector per track, 100–199 may have two per track, etc. 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 4096 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 interblock gap. Table 13.1 represents specifications of a typical disk.

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 0.1 cent/MB which translates to $1/GB and $1K/TB are not too far away.

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 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 reserved area in main storage that holds one block—is also provided. For a **read** command, the block from disk is copied into the buffer; whereas for a **write** command, the contents of the buffer are copied into the disk block.

TABLE 13.1 SPECIFICATIONS OF TYPICAL HIGH-END CHEETAH DISKS FROM SEAGATE

Description	Cheetah X15 36LP	Cheetah 10K.6
Model Number	ST336732LC	ST3146807LC
Form Factor (width)	3.5 inch	3.5 inch
Height	25.4 mm	25.4 mm
Width	101.6 mm	101.6 mm
Weight	0.68 Kg	0.73 Kg
Capacity/Interface		
Formatted Capacity	36.7 Gbytes	146.8 Gbytes
Interface Type	80-pin	80-pin
Configuration		
Number of disks (physical)	4	4
Number of heads (physical)	8	8
Number of Cylinders	18,479	49,854
Bytes per Sector	512	512
Areal Density	N/A	36,000 Mbits/sq.inch
Track Density	N/A	64,000 Tracks/inch
Recording Density	N/A	570,000 bits/inch
Performance		
Transfer Rates		
Internal Transfer Rate (min)	522 Mbits/sec	475 Mbits/sec
Internal Transfer Rate (max)	709 Mbits/sec	840 Mbits/sec
Formatted Int. Transfer Rate (min)	51 MBytes/sec	43 MBytes/sec
Formatted Int. Transfer Rate (max)	69 MBytes/sec	78 MBytes/sec
External I/O Transfer Rate (max)	320 MBytes/sec	320 MBytes/sec
Seek Times		
Avg. Seek Time (Read)	3.6 msec (typical)	4.7 msec (typical)
Avg. Seek Time (Write)	4.2 msec (typical)	5.2 msec (typical)
Track-to-track Seek, Read	0.5 msec (typical)	0.3 msec (typical)
Track-to-track Seek, Write	0.8 msec (typical)	0.5 msec (typical)
Average Latency	2 msec	2.99 msec
Other		
Default Buffer (cache) size	8,192 Kbytes	8,000 Kbytes
Spindle Speed	15K rpm	10K rpm

TABLE 13.1 SPECIFICATIONS OF TYPICAL HIGH-END CHEETAH DISKS FROM SEAGATE (continued)

Reliability		
Mean Time Between Failure (MTBF)	1,200,000 Hours	1,200,000 Hours
Recoverable Read Errors	10 per 10^{12} bits	10 per 10^{12} bits
Nonrecoverable Read Errors	1 per 10^{15} bits	1 per 10^{15} bits
Seek Errors	10 per 10^{8} bits	10 per 10^{8} bits

(courtesy Seagate Technology)

Sometimes several contiguous blocks, called a **cluster,** may be transferred as a unit. In this case the buffer size is adjusted to match the number of bytes in the cluster.

The actual hardware mechanism that reads or writes a block is the disk **read/write head,** which is part of a system called a **disk drive.** A disk or disk pack is mounted in the disk drive, which includes a motor that rotates the disks. A read/write head includes an electronic component attached to a **mechanical arm.** Disk packs with multiple surfaces are controlled by several read/write heads—one for each surface (see Figure 13.1b). 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 5400 and 15,000 rpm). For a floppy disk, the disk drive begins to rotate the disk whenever a particular read or write request is initiated and ceases rotation soon after the data transfer is completed. Once the read/write head is positioned on the right track and the block specified in the block address moves under the read/write head, the electronic component of the read/write head is activated to transfer the data. Some disk units have fixed read/write heads, with as many heads as there are tracks. These are called **fixed-head** disks, whereas disk units with an actuator are called **movable-head disks.** For fixed-head disks, a track or cylinder is selected by electronically switching to the appropriate read/write head rather than by actual mechanical movement; consequently, it is much faster. However, the cost of the additional read/write heads is quite high, so fixed-head disks are not commonly used.

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 PC and workstations is called SCSI (Small Computer Storage 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 7 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. 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 12 to 60 msec. For contiguous blocks, locating the first block takes from 12 to 60 msec, but transferring subsequent blocks may take only 1 to 2 msec each. Many search techniques take advantage of consecutive retrieval of blocks when searching for data on disk. In any case, a transfer time in the order of milliseconds is considered quite high compared 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 14 attempt to *minimize the number of block transfers* needed to locate and transfer the required data from disk to main memory.

13.2.2 Magnetic Tape Storage Devices

Disks are **random access** secondary storage devices, because an arbitrary disk block may be accessed "at random" once we specify its address. Magnetic tapes are **sequential access** devices; to access the n^{th} block on tape, we must first scan over the preceding $n - 1$ blocks. Data is stored on reels of high-capacity magnetic tape, somewhat similar to audio- or videotapes. A **tape drive** is required to read the data from or to write the data to a **tape reel.** Usually, each group of bits that 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[5] of 0.6 inches corresponds to 960 to 3750 bytes of wasted storage space. For better space utilization it is customary to group many records together in one block.

The main characteristic of a tape is its requirement that we access the data blocks in **sequential order.** To get to a block in the middle of a reel of tape, 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—that of **backing up** the database. One reason for backup is to keep copies of disk files in case the data is lost because of a disk crash, which can happen if the disk read/write head touches the disk surface because of mechanical malfunction. For this reason, disk files are copied periodically to tape. For many online critical applications such as airline reservation

5. Called *interrecord gaps* in tape terminology.

systems, to avoid any downtime, mirrored systems are used keeping three sets of identical disks—two in online operation and one 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. Finally, database files that are seldom used or are outdated but are required for historical record keeping can be **archived** on tape. Recently, smaller 8-mm magnetic tapes (similar to those used in camcorders) that can store up to 50 Gbytes, as well as 4-mm helical scan data cartridges and writable CDs and DVDs have become popular media for backing up data files from workstations and personal computers. 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 L5500 model of Storage Technology that can scale up to 13.2 Petabytes (Petabyte = 1000 TB) with a thruput rate of 55TB/hour. We defer the discussion of disk storage technology called RAID, and of storage area networks, to the end of the chapter.

13.3 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. This is possible 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 13.3 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 13.4 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 write a continuous stream of blocks from memory to the disk. Double buffering permits continuous reading or writing of data on consecutive disk blocks, which eliminates the seek time and rotational delay for all but the first block transfer. Moreover, data is kept ready for processing, thus reducing the waiting time in the programs.

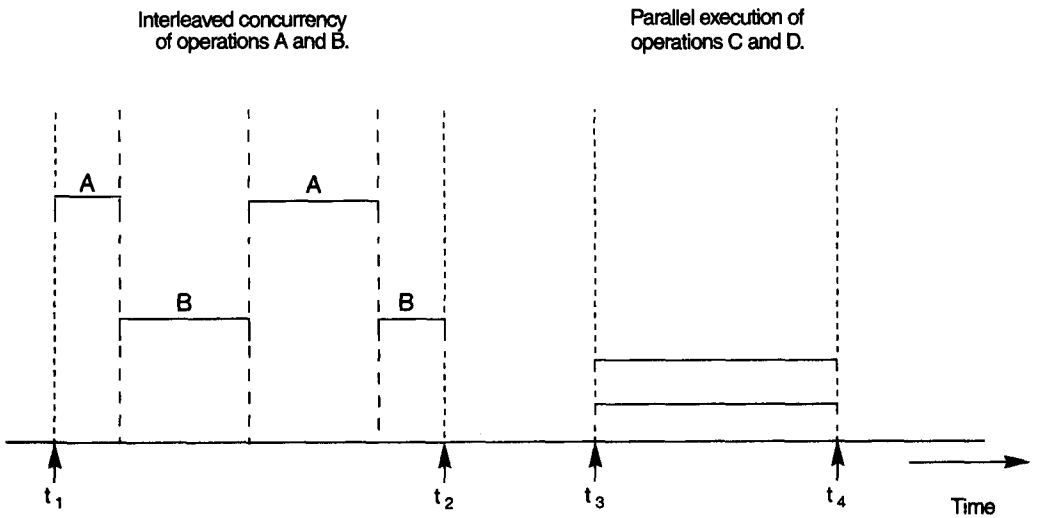


FIGURE 13.3 Interleaved concurrency versus parallel execution.

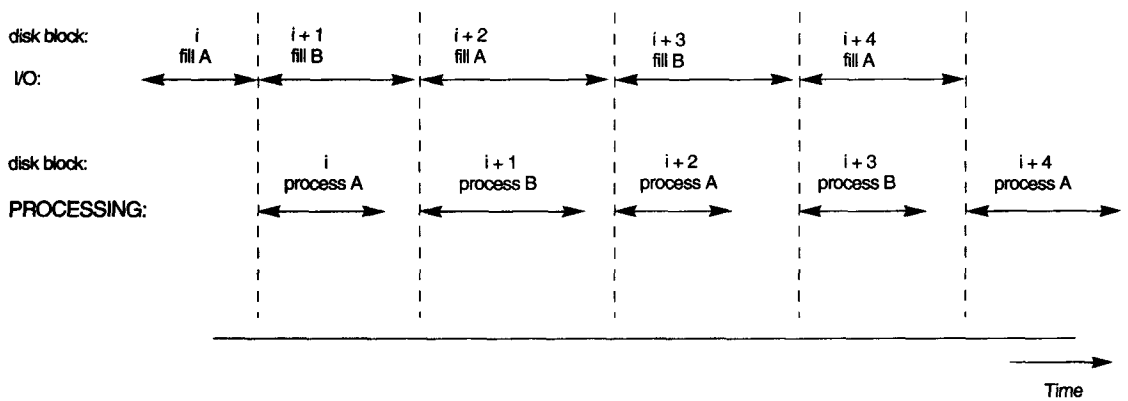


FIGURE 13.4 Use of two buffers, A and B, for reading from disk.

13.4 Placing File Records on Disk

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

13.4.1 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 corre-

sponds 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, BIRTHDATE, SALARY, or SUPERVISOR. A collection of field names and their corresponding data types constitutes a **record type** or **record format** definition. A **data type,** associated with each field, specifies the 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 jobcode;
   char department[20];
};
```

In recent database applications, the need may arise for storing data items that consist of large unstructured objects, which represent images, digitized video or audio streams, or free text. These are referred to as **BLOBs** (Binary Large Objects). 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.

13.4.2 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 13.5a 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

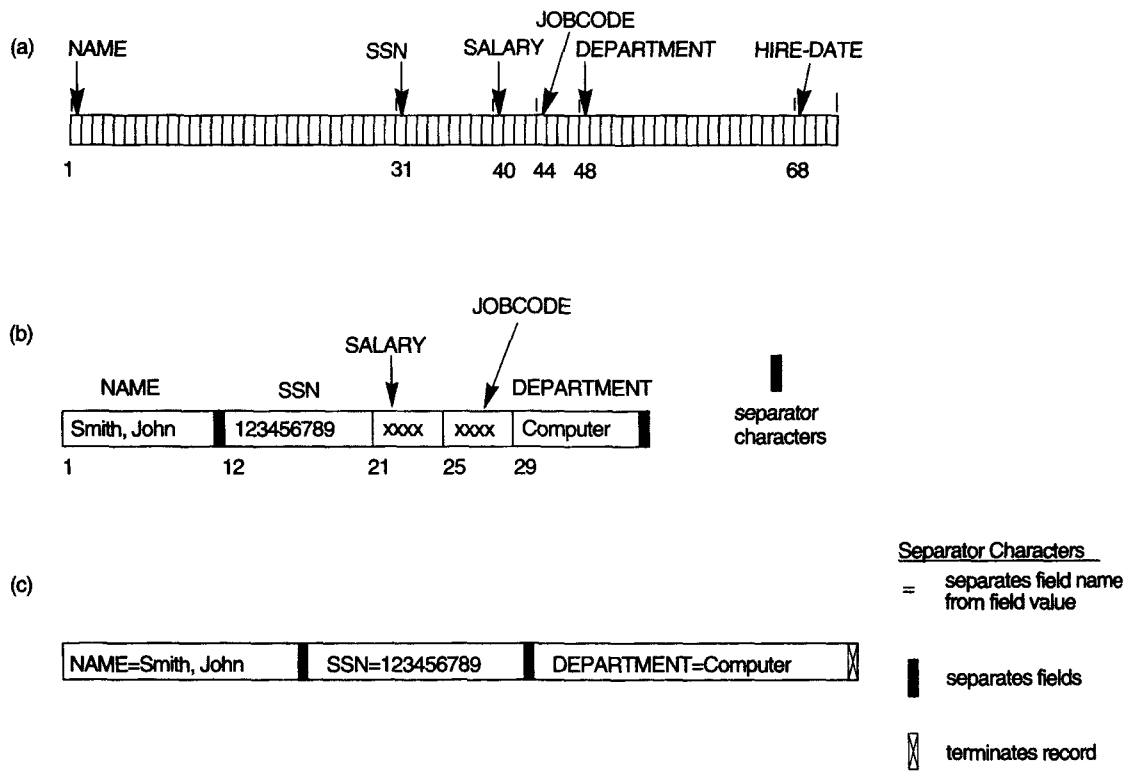


FIGURE 13.5 Three record storage formats. (a) A fixed-length record with six fields and size of 71 bytes. (b) A record with two variable-length fields and three fixed-length fields. (c) A variable-field record with three types of separator characters.

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

For *variable-length fields*, each record has a value for each field, but we do not know the exact length of some field values. To determine the bytes within a particular record that represent each field, we can use special **separator** characters (such as ? or % or $)—which do not appear in any field value—to terminate variable-length fields (Figure 13.5b), 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 13.7c, 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.[6]

13.4.3 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

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

6. Other schemes are also possible for representing variable-length records.

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 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 13.6 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.

13.4.4 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

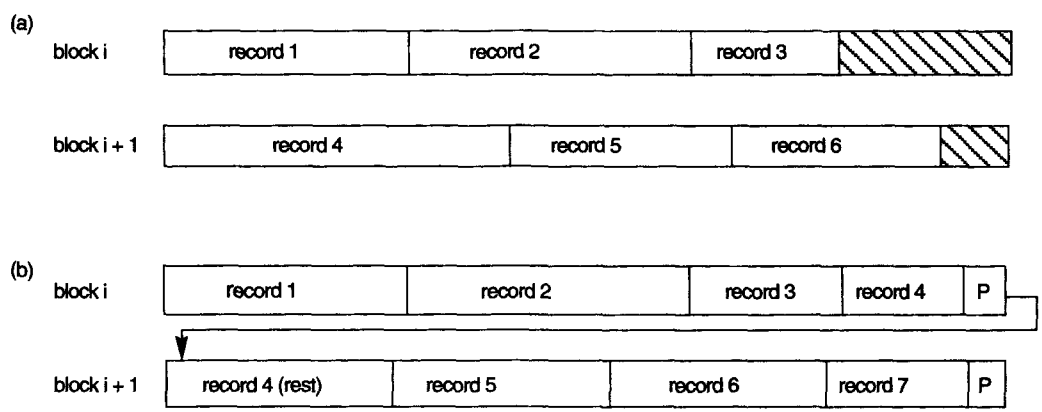


FIGURE 13.6 Types of record organization. (a) Unspanned. (b) Spanned.

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.

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

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

Search operations on 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 the 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.
- *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 mamory 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 6).

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 (Figure 13.5a), 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 (say, 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 6). Suppose that a second application uses the file to generate employees' paychecks and requires that paychecks be grouped by department. For this application, it is best to store all employee records having the same department value contiguously, clustering them into blocks and perhaps ordering them by name within each department. 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 there may not be an organization that allows all needed operations on a file to be implemented efficiently. In such cases 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 6, 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. In addition, various general techniques for handling insertions and deletions work with many file organizations.

13.6 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.**[7] This organization is often used with additional access paths, such as the secondary indexes discussed in Chapter 6. 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, then 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 of the marker indicates a valid (not deleted) record. Search programs consider only valid records in a block when conducting their search. Both of these deletion techniques require periodic **reorganization** of the file to reclaim the unused space of deleted records. During reorganization, the file blocks are accessed consecutively, and records are packed by removing deleted records. After such a reorganization, the blocks are filled to capacity once more. Another possibility is to use the space of deleted records when inserting new records, although this requires extra bookkeeping to keep track of empty locations.

We can use either spanned or unspanned organization for an unordered file, and 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 15).

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 $0, 1, 2, \ldots, r - 1$ and the records in each block are numbered $0, 1, \ldots, bfr - 1$, where bfr is the blocking factor, then the i^{th} record of the file is located in block $\lfloor (i/bfr) \rfloor$ and is the $(i \bmod bfr)^{th}$ 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

7. Sometimes this organization is called a **sequential file.**

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 6.

13.7 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.[8] 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 13.7 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.

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 13.1. 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 13.1: 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 2 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;
```

8. The term *sequential file* has also been used to refer to unordered files.

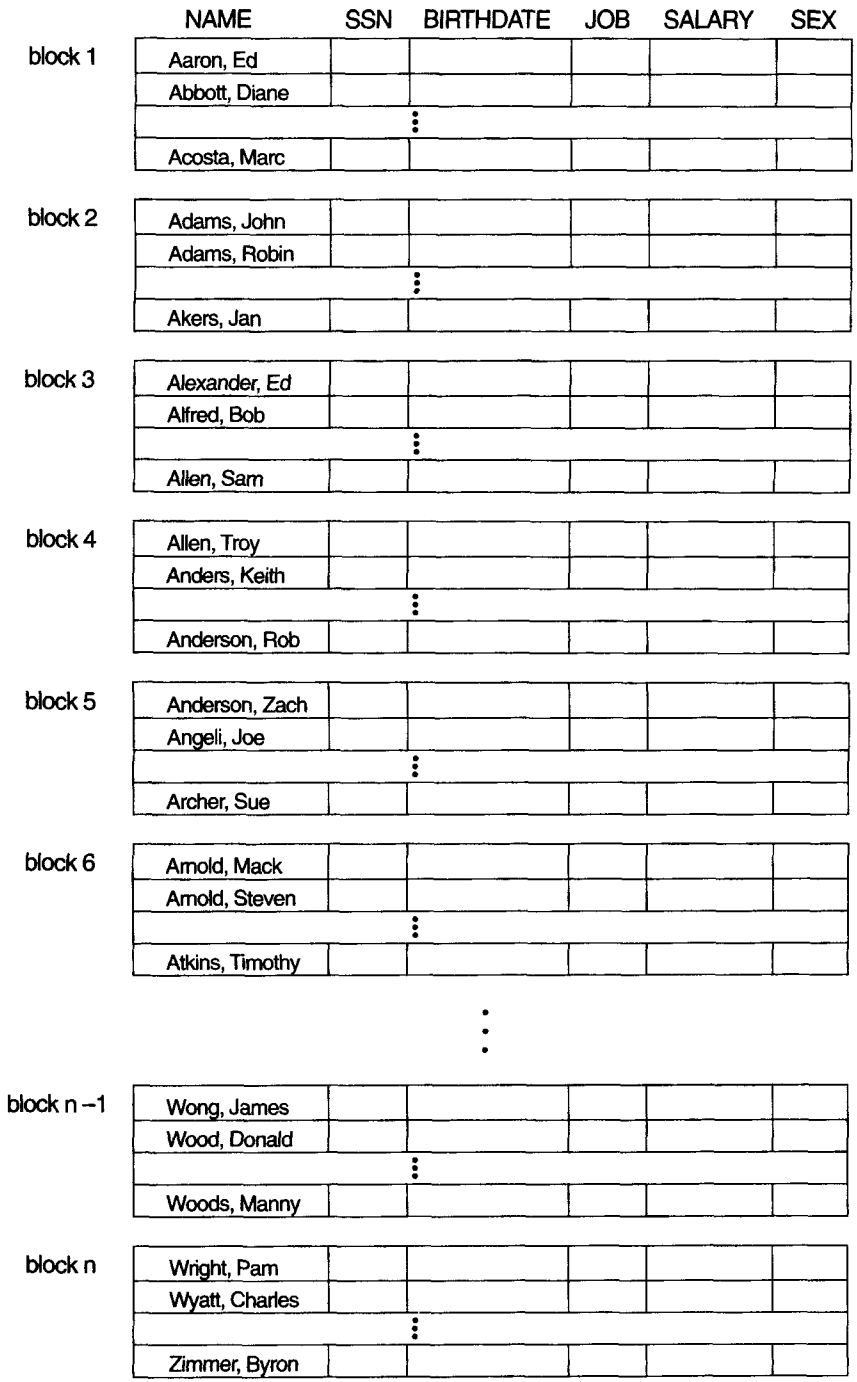


FIGURE 13.7 Some blocks of an ordered (sequential) file of EMPLOYEE records with NAME as the ordering key field.

A search criterion involving the conditions >, <, ≥, and ≤ on the ordering field is quite efficient, since the physical ordering of records means that all records satisfying the condition are contiguous in the file. For example, referring to Figure 13.9, 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.

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: (1) the search condition to locate the record and (2) 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, which requires deletion of the old record followed by insertion of the modified record.

Reading the file records in order of the ordering field is quite efficient if we ignore the records in overflow, since the blocks can be read consecutively using double buffering. To include the records in overflow, we must merge them in their correct positions; in this case, we can first reorganize the file, and then read its blocks sequentially. To reorganize the file, first 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 13.2 AVERAGE ACCESS TIMES FOR BASIC FILE ORGANIZATIONS

TYPE OF ORGANIZATION	ACCESS/SEARCH METHOD	AVERAGE TIME TO ACCESS A SPECIFIC RECORD
Heap (Unordered)	Sequential scan (Linear Search)	$b/2$
Ordered	Sequential scan	$b/2$
Ordered	Binary Search	$\log_2 b$

Table 13.2 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 further improves the random access time on the ordering key field. We discuss indexes in Chapter 14.

13.8 HASHING TECHNIQUES

Another type of primary file organization is based on hashing, which provides very fast access to records on certain search conditions. This organization is usually called a **hash file.**[9] The search condition must be an equality condition on a single field, called the **hash field** of the file. 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,** that is applied to the hash field value of a record and yields the *address* of the disk block in which the record is stored. A search for the record within the block can be carried out in a main memory buffer. For most records, we need only a single-block access to retrieve that record.

Hashing is also used as an internal 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 13.9.1; then we show how it is modified to store external files on disk in Section 13.9.2. In Section 13.9.3 we discuss techniques for extending hashing to dynamically growing files.

13.8.1 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 (Figure 13.8a); 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

9. A hash file has also been called a *direct file*.

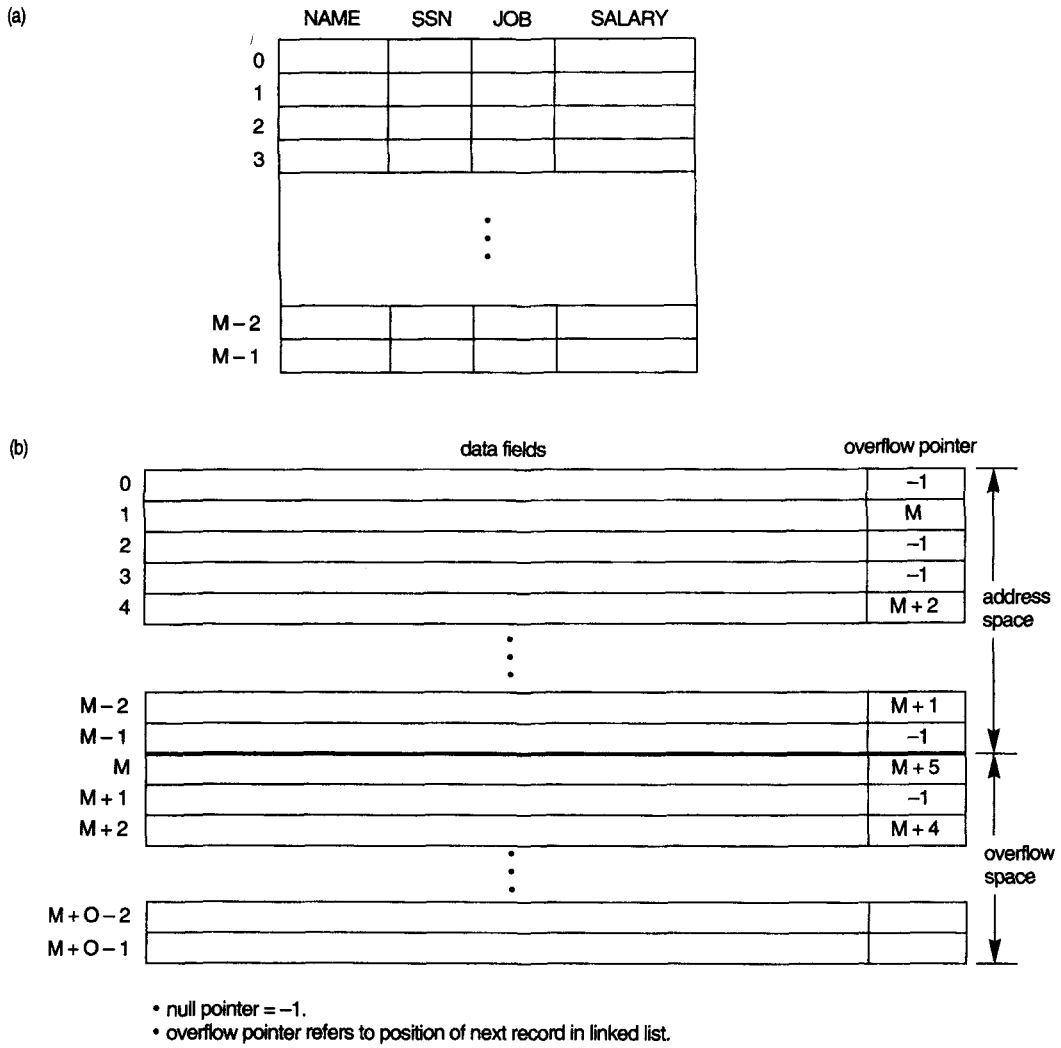


FIGURE 13.8 Internal hashing data structures. (a) Array of M positions for use in internal hashing. (b) Collision resolution by chaining records.

an integer hash field value K after division by M; this value is then used for the record address.

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 13.2a 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 13.2 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 fi 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. Another technique involves picking some digits of the hash field value—for example, the third, fifth, and eighth digits—to form the hash address.[10] 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 13.2b may be used for this purpose.
- *Chaining:* For this method, various overflow locations are kept, usually by extending the array with a number of overflow positions. In addition, a pointer field is added to each record location. A collision is resolved by placing the new record in an unused overflow location and setting the pointer of the occupied hash address location to the address of that overflow location. A linked list of overflow records for each hash address is thus maintained, as shown in Figure 13.8b.
- *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.

10. A detailed discussion of hashing functions is outside the scope of our presentation.

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.

13.8.2 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 blocks. The hashing function maps a key into a relative bucket number, rather than assign 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 13.9.

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

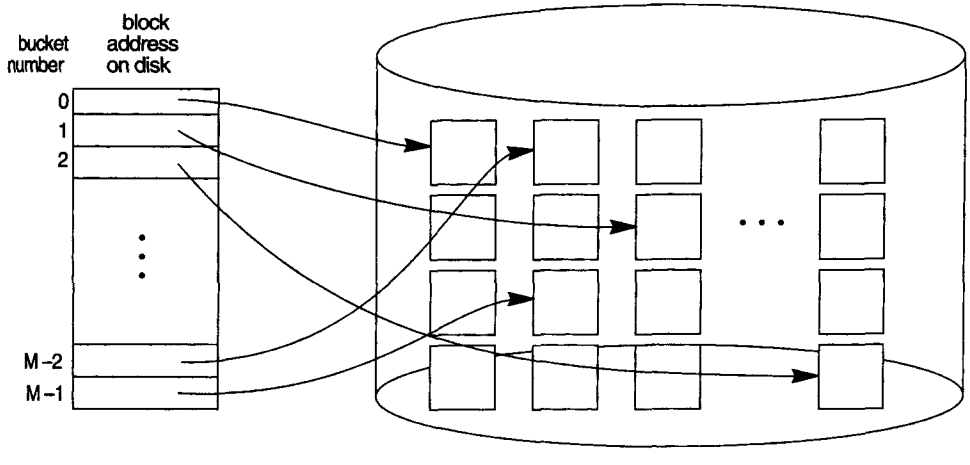


FIGURE 13.9 Matching bucket numbers to disk block addresses.

Figure 13.10. 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 as the hash address and keep the records sorted by invoice number within each bucket. Another example is to use an integer hash key directly as an index to a relative file, if the hash key values fill up a particular interval; for example, if employee numbers in a company are assigned as 1, 2, 3, . . . up to the total number of employees, we can use the identity hash function that maintains order. Unfortunately, this only works if keys are generated in order by some application.

The hashing scheme described 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

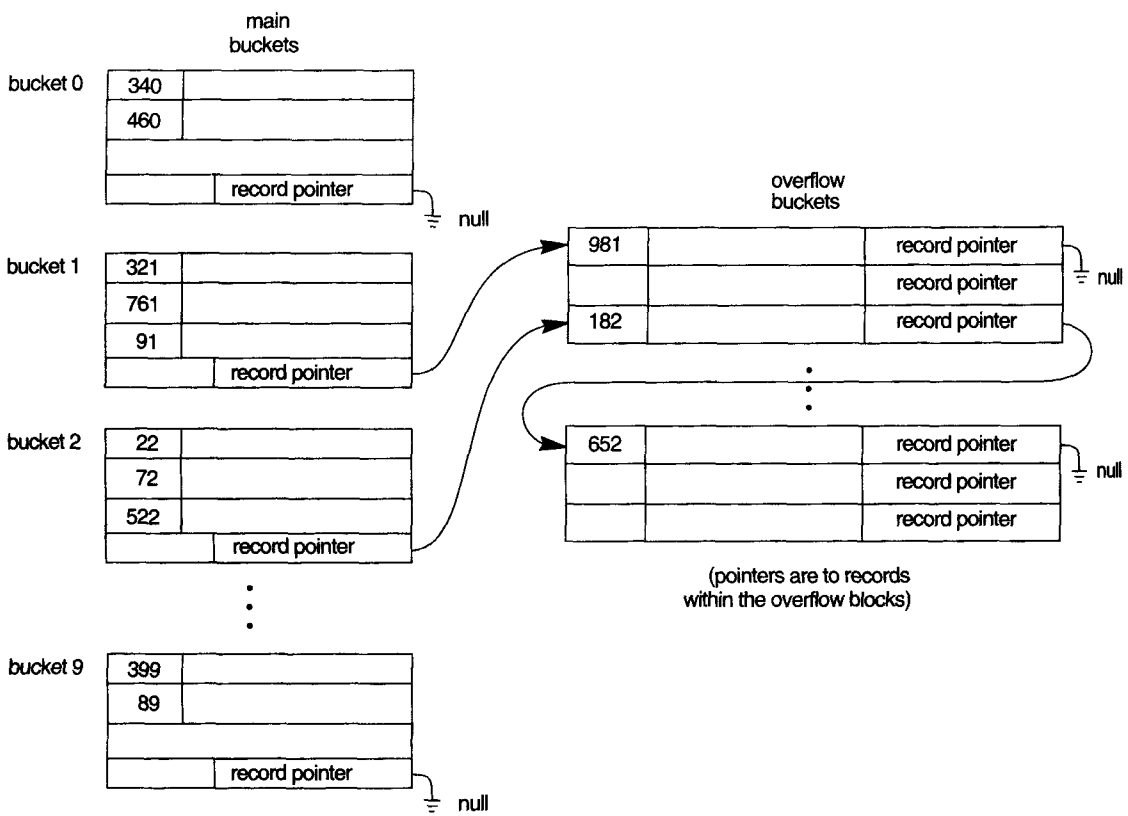


FIGURE 13.10 Handling overflow for buckets by chaining.

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 13.8.3).

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.

Modifying a record's field value depends on two factors: (1) the search condition to locate the record and (2) 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.

13.8.3 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 (Chapter 6). 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.

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

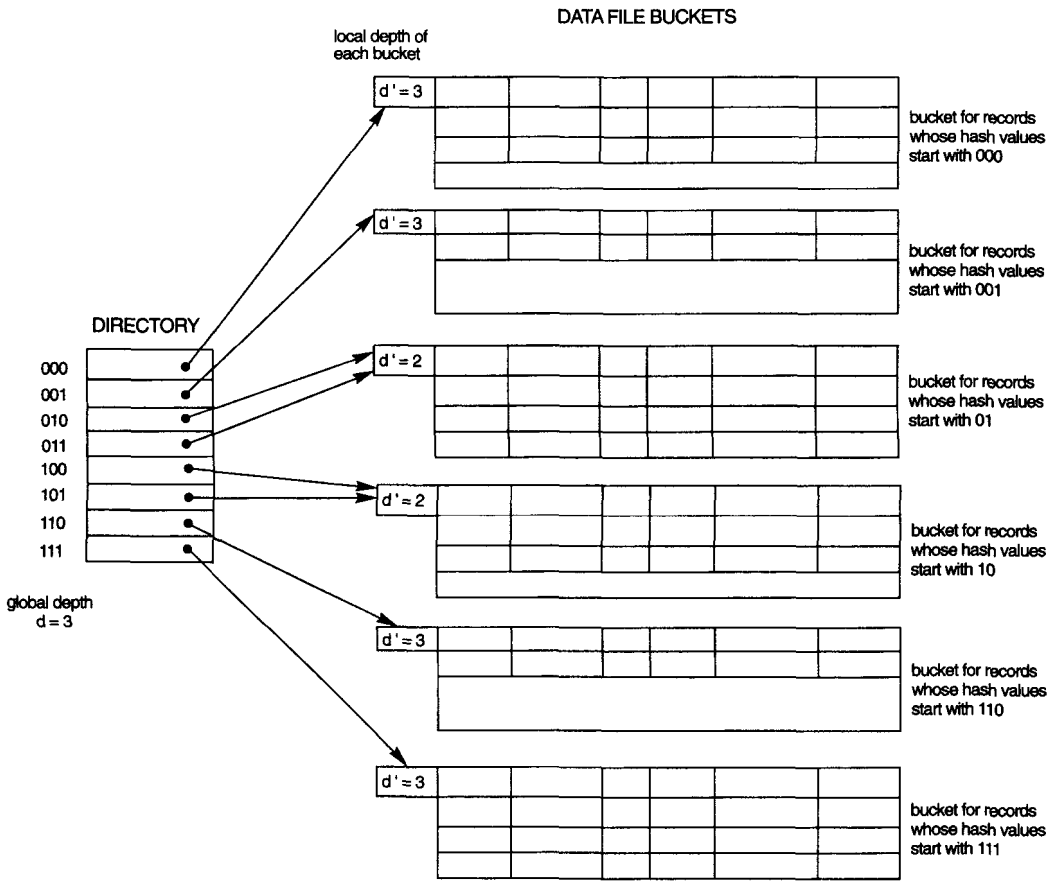


FIGURE 13.11 *Structure of the extendible hashing scheme.*

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 13.13 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 13.13. 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 13.11 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 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 causes additional accesses. In addition, 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 a 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 hence the scheme is considered quite desirable for dynamic files.

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 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 $h_{i+j}(K) = K \bmod (2^jM)$ is used, where $j = 0, 1, 2, \ldots$; a new hashing function h_{i+j+1} is needed whenever all the buckets $0, 1, \ldots, (2^jM) - 1$ have been split and n is reset to 0. The search for a record with hash key value K is given by Algorithm 13.3.

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 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.

Algorithm 13.3: The search procedure for linear hashing.

```
if n = 0
    then m ← hj(K) (* m is the hash value of record with hash key K *)
    else begin
       m ← hj(K);
       if m < n then m ← hj+1 (K)
       end;
search the bucket whose hash value is m (and its overflow, if any);
```

13.9 Other Primary File Organizations

13.9.1 Files of Mixed Records

The file organizations we have studied so far assume that all records of a particular file are of the same record type. The records could be of EMPLOYEES, PROJECTS, STUDENTS, or DEPARTMENTS, but each file contains records of only one type. In most database applications, we encounter situations in which numerous types of entities are interrelated in various ways, as we saw in Chapter 3. Relationships among records in various files can be represented by **connecting fields.**[11] For example, a STUDENT record can have a connecting field MAJORDEPT whose

11. The concept of foreign keys in the relational model (Chapter 5) and references among objects in object-oriented models (Chapter 20) are examples of connecting fields.

value gives the name of the DEPARTMENT in which the student is majoring. This MAJORDEPT field *refers* to a DEPARTMENT entity, which should be represented by a record of its own in the DEPARTMENT file. If we want to retrieve field values from two related records, we must retrieve one of the records first. Then we can use its connecting field value to retrieve the related record in the other file. Hence, relationships are implemented by **logical field references** among the records in distinct files.

File organizations in 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 very frequently, implementing the relationship physically can increase the system's efficiency at retrieving related records. For example, if the query to retrieve a DEPARTMENT record and all records for STUDENTS majoring in that department is very frequent, it would be desirable to place each DEPARTMENT record and its cluster of STUDENT records contiguously on disk in a mixed file. 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.

13.9.2 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 14.3.1, 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.

13.10 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 in performance and reliability with processor technology.

A major advance in secondary storage technology is represented by the development of RAID, which originally stood for **Redundant Arrays of Inexpensive Disks.** Lately, the "I" in RAID is said to stand for Independent. The RAID idea received a very positive endorsement by industry 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 below.

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. Table 13.3 shows trends in disk technology in terms of 1993 parameter values and rates of improvement, as well as where these parameters are in 2003.

A second qualitative disparity exists between the ability of special microprocessors that cater to new applications involving processing of video, audio, image, and spatial data (see Chapters 24 and 29 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 13.12 shows a file distributed or *striped* over four disks. Striping improves overall I/O performance by

TABLE 13.3 TRENDS IN DISK TECHNOLOGY

	1993 PARAMETER VALUES*	HISTORICAL RATE OF IMPROVEMENT PER YEAR (%)*	CURRENT (2003) VALUES**
Areal density	50–150 Mbits/sq. inch	27	36 Gbits/sq. inch
Linear density	40,000–60,000 bits/inch	13	570 Kbits/inch
Inter-track density	1500–3000 tracks/inch	10	64,000 tracks/inch
Capacity (3.5" form factor)	100–2000 MB	27	146 GB
Transfer rate	3–4 MB/s	22	43–78 MB/sec
Seek time	7–20 ms	8	3.5–6 msec

*Source: From Chen, Lee, Gibson, Katz, and Patterson (1994), ACM *Computing Surveys*, Vol. 26, No. 2 (June 1994). Reprinted by permission.

**Source: IBM Ultrastar 36XP and 18ZX hard disk drives.

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.

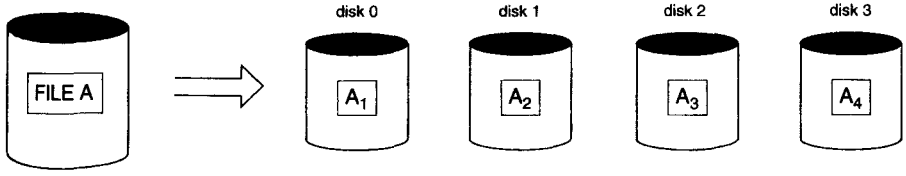


FIGURE 13.12 Data striping. File A is striped across four disks.

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 13.3.1 and 13.3.2, we discuss how RAID achieves the two important objectives of improved reliability and higher performance. Section 13.3.3 discusses RAID organizations.

13.10.1 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 MTTF (Mean Time To Failure) of a disk drive is assumed to be 200,000 hours or about 22.8 years (typical times range up to 1 million hours), that of a bank of 100 disk drives becomes only 2000 hours or 83.3 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 MTTF 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: (1) selecting a technique for computing the redundant information, and (2) 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

13. The formulas for MTTF calculations appear in Chen et al. (1994).

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 hence to improve reliability.

13.10.2 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 bytes. Disk striping may be applied at a finer granularity by 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).

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 13.12 shows block-level data striping assuming the data file contained 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 a $1/100^{th}$ the reliability of a single disk. Thus, redundancy via error-correcting codes and disk mirroring is necessary to provide reliability along with high performance.

13.10.3 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. 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. 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. The seven RAID levels (0 through 6) are illustrated in Figure 13.13 schematically.

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. 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.

13.11 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 27), 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, large organizations are moving 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-

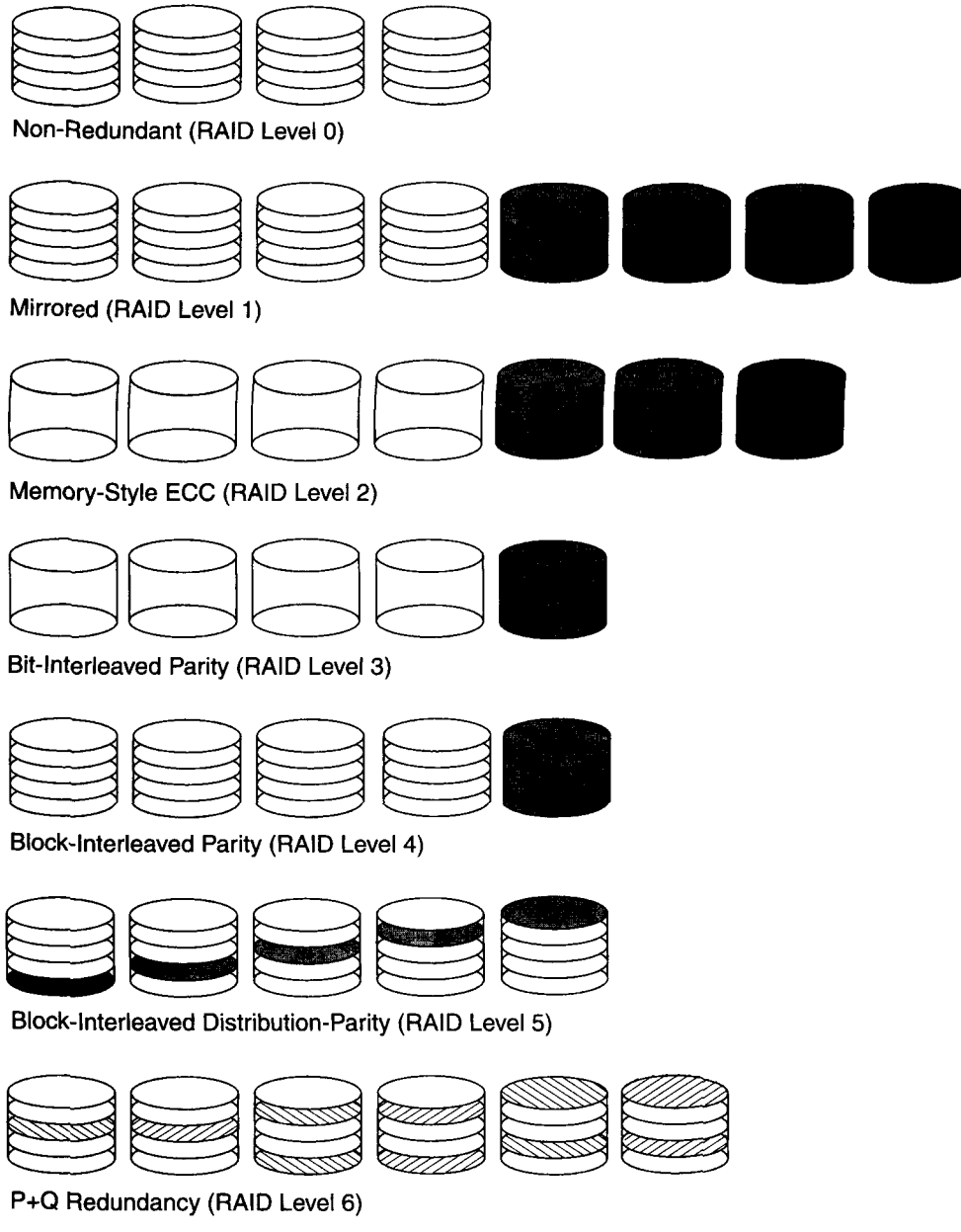


FIGURE 13.13 Multiple levels of RAID. From Chen, Lee, Gibson, Katz, and Patterson (1994), ACM Computing Survey, Vol. 26, No. 2 (June 1994). Reprinted with permisson.

switch to connect multiple RAID systems, tape libraries, etc. to servers, 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 vendors of SANs. The main advantages claimed are the following:

- 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 SAN as a viable option for database storage.

13.12 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. Double buffering can be used when accessing consecutive disk blocks, to reduce the average block access time. Other disk parameters are discussed in Appendix B. We presented different ways of storing records of a file on disk. Records of a file are grouped into disk blocks and can be of fixed length or variable length, spanned or unspanned, and 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.

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

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

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

Hashing provides very fast access to an arbitrary record of a file, given the value of its hash 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 then discussed two hashing techniques for files that grow and shrink in the number of records dynamically—namely, extendible and linear hashing.

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. Finally, we reviewed the recent advances in disk technology represented by RAID (Redundant Arrays of Inexpensive [Independent] Disks).

Review Questions

13.1. What is the difference between primary and secondary storage?
13.2. Why are disks, not tapes, used to store online database files?
13.3. Define the following terms: *disk*, *disk pack*, *track*, *block*, *cylinder*, *sector*, *interblock gap*, *read/write head*.
13.4. Discuss the process of disk initialization.
13.5. Discuss the mechanism used to read data from or write data to the disk.
13.6. What are the components of a disk block address?
13.7. Why is accessing a disk block expensive? Discuss the time components involved in accessing a disk block.
13.8. Describe the mismatch between processor and disk technologies.
13.9. What are the main goals of the RAID technology? How does it achieve them?
13.10. How does disk mirroring help improve reliability? Give a quantitative example.
13.11. What are the techniques used to improve performance of disks in RAID?
13.12. What characterizes the levels in RAID organization?
13.13. How does double buffering improve block access time?
13.14. What are the reasons for having variable-length records? What types of separator characters are needed for each?
13.15. Discuss the techniques for allocating file blocks on disk.
13.16. What is the difference between a file organization and an access method?
13.17. What is the difference between static and dynamic files?
13.18. What are the typical record-at-a-time operations for accessing a file? Which of these depend on the current record of a file?
13.19. Discuss the techniques for record deletion.
13.20. 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?
13.21. Discuss the techniques for allowing a hash file to expand and shrink dynamically. What are the advantages and disadvantages of each?
13.22. What are mixed files used for? What are other types of primary file organizations?

Exercises

13.23. 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.

13.24. A file has $r = 20{,}000$ STUDENT records of *fixed length*. Each record has the following fields: NAME (30 bytes), SSN (9 bytes), ADDRESS (40 bytes), PHONE (9 bytes), BIRTHDATE (8 bytes), SEX (1 byte), MAJORDEPTCODE (4 bytes), MINORDEPTCODE (4 bytes), CLASSCODE (4 bytes, integer), and DEGREEPROGRAM (3 bytes). An additional byte is used as a deletion marker. The file is stored on the disk whose parameters are given in Exercise 13.23.

a. Calculate the record size R in bytes.

b. Calculate the blocking factor *bfr* and the number of file blocks b, assuming an unspanned organization.

c. Calculate the average time it takes to find a record by doing a linear search on the file if (i) the file blocks are stored contiguously, and double buffering is used; (ii) the file blocks are not stored contiguously.

d. Assume that the file is ordered by SSN; calculate the time it takes to search for a record given its SSN value, by doing a binary search.

13.25. Suppose that only 80 percent of the STUDENT records from Exercise 13.24 have a value for PHONE, 85 percent for MAJORDEPTCODE, 15 percent for MINORDEPTCODE, and 90 percent for DEGREEPROGRAM; 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.

13.26. Suppose that a disk unit has the following parameters: seek time s = 20 msec; rotational delay rd = 10 msec; block transfer time btt = 1 msec; block size B = 2400 bytes; interblock gap size G = 600 bytes. An EMPLOYEE file has the following fields: SSN, 9 bytes; LASTNAME, 20 bytes; FIRSTNAME, 20 bytes; MIDDLE INIT, 1 byte; BIRTHDATE, 10 bytes; ADDRESS, 35 bytes; PHONE, 12 bytes; SUPERVISORSSN, 9 bytes; DEPARTMENT, 4 bytes; JOBCODE, 4 bytes; *deletion marker*, 1 byte. The EMPLOYEE file has r = 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.

13.27. A PARTS file with Part# as hash key includes records with the following Part# values: 2369, 3760, 4692, 4871, 5659, 1821, 1074, 7115, 1620, 2428, 3943, 4750, 6975, 4981, 9208. The file uses eight buckets, numbered 0 to 7. Each bucket is one disk block and holds two records. Load these records into the file in the given order, using the hash function $h(K) = K \bmod 8$. Calculate the average number of block accesses for a random retrieval on Part#.

13.28. Load the records of Exercise 13.27 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 $h(K) = K \bmod 128$.

13.29. Load the records of Exercise 13.27 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.

13.30. Compare the file commands listed in Section 13.6 to those available on a file access method you are familiar with.

13.31. 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.

13.32. 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.

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

13.34. 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.

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

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

13.37. 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 forth. 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.

13.38. 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?

13.39. 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.

13.40. 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 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, btt = 1 ms.

13.41. 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 (1983) has a detailed discussion and analysis of secondary storage devices and file organizations. Optical disks are described in Berg and Roth (1989) and analyzed in Ford and Christodoulakis 1991. Flash memory is discussed by Dippert 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 (1973), discuss static hashing in more detail; Knuth has a complete discussion of hash functions and collision resolution techniques, as well as of their performance comparison. Knuth also offers a detailed discussion of techniques for sorting external files. Textbooks on file structures include Claybrook (1983), Smith and Barnes (1987), and Salzberg (1988); they discuss additional file organizations including tree structured files, and have detailed algorithms for operations on files. Additional textbooks on file organizations include Miller (1987), and Livadas (1989). 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 is 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). Dynamic hashing, which we did not discuss in detail, 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; e.g., www.seagate.com, www.ibm.com, www.storagetek.com. IBM has a storage technology research center at IBM Almaden (www.almaden.ibm.com/sst/).

14

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 13. 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 typically provide **secondary access paths,** which provide alternative ways of accessing the records without affecting the physical placement of records 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 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 file based on a certain selection criterion on an indexing field, one has to initially access the index, which points to one or more blocks in the 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 describe different types of single-level ordered indexes—primary, secondary, and clustering—in Section 14.1. 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 indexes in Section 14.2. In Section 14.3 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 14.4 is devoted to the alternative ways of accessing data based on a combination of multiple keys. In Section 14.5, we discuss how other data structures—such as hashing—can be used to construct indexes. We also briefly 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. Section 14.6 summarizes the chapter.

14.1 Types of Single-Level Ordered Indexes

The idea behind an ordered index access structure 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 an index to find a list of *addresses*—page numbers in this case—and use these addresses to locate a term in the textbook by *searching* the specified pages. 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 on a file. Of course, most books do have additional information, such as chapter and section titles, that can help us find a term without having to search through the whole book. However, the index is the only exact indication of 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. The index file is much smaller than the data file, so searching the index using a binary search is reasonably efficient. Multilevel indexing (see Section 14.2) does away with the need for a binary search at the expense of creating indexes to the index itself.

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

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

14.1.1 Primary Indexes

A **primary index** is an ordered file whose records are of fixed length with two fields. 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 13.7, 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) = (Aaron,Ed), P(1) = address of block 1>

<K(2) = (Adams,John), P(2) = address of block 2>

<K(3) = (Alexander,Ed), P(3) = address of block 3>

Figure 14.1 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 primary index is hence 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 needs substantially fewer blocks 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. A binary search on the index file hence requires fewer block accesses than a binary search on the data file. Referring back to Table 13.2, note that the binary search for an ordered data file required $\log_2 b$ block accesses. But if the primary index file contains b_i blocks, then to locate a record with a search key 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 i^{th} 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]

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.

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.

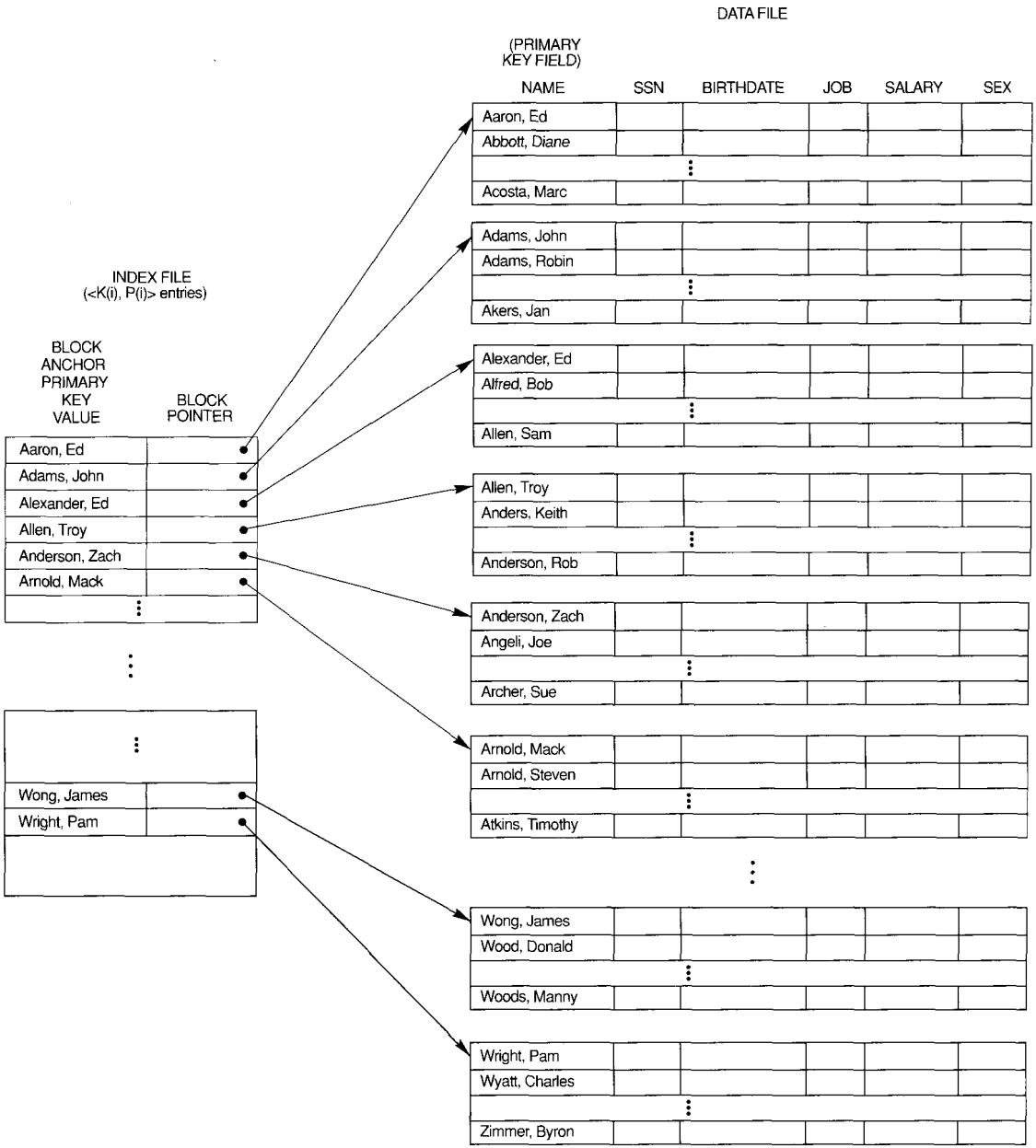


FIGURE 14.1 Primary index on the ordering key field of the file shown in Figure 13.7.

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 = $\lceil (r/bfr) \rceil$ = $\lceil (30{,}000/10) \rceil$ = 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 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 have to 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 13.7, 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 13.8.2. Records within each block and its overflow linked list can be sorted to improve retrieval time. Record deletion is handled using deletion markers.

14.1.2 Clustering Indexes

If records of a file are physically ordered on a nonkey field—which *does not* have a distinct value for each record—that field is called the **clustering field.** We can create a different type of index, called a **clustering index,** to speed up retrieval of 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 block pointer. There is one entry in the clustering index for each *distinct value* of the clustering field, containing 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 14.2 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

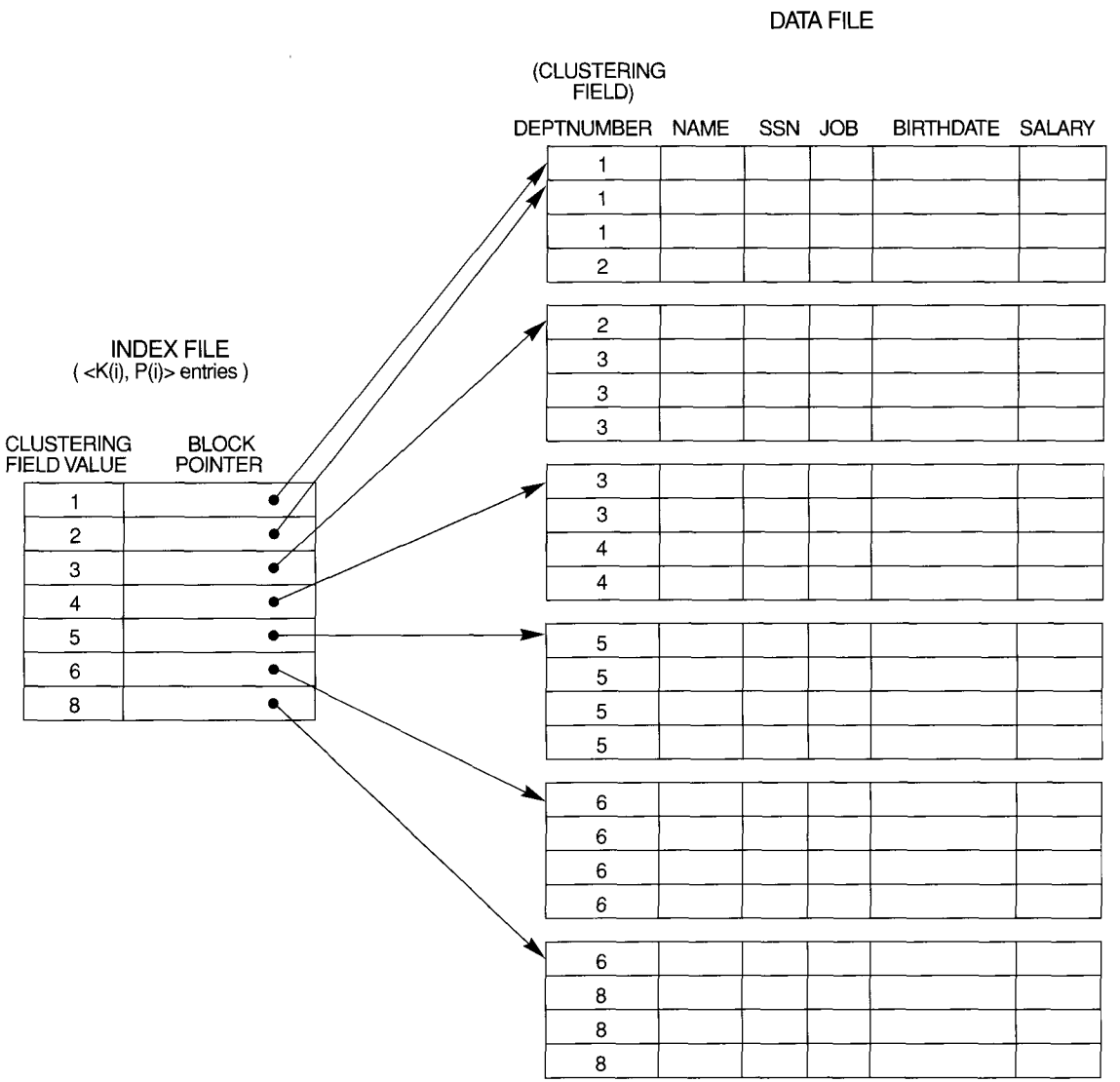


FIGURE 14.2 A clustering index on the DEPTNUMBER ordering nonkey field of an EMPLOYEE file.

block (or block cluster). This makes insertion and deletion relatively straightforward. Figure 14.3 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 for every record in the file. There is some similarity between Figures 14.1 to 14.3, on the one hand, and Figure 13.11, on the other. An index is somewhat similar to the directory structures used for extendible hashing, described in Section 13.8.3. Both are searched to find a pointer to the data block containing the

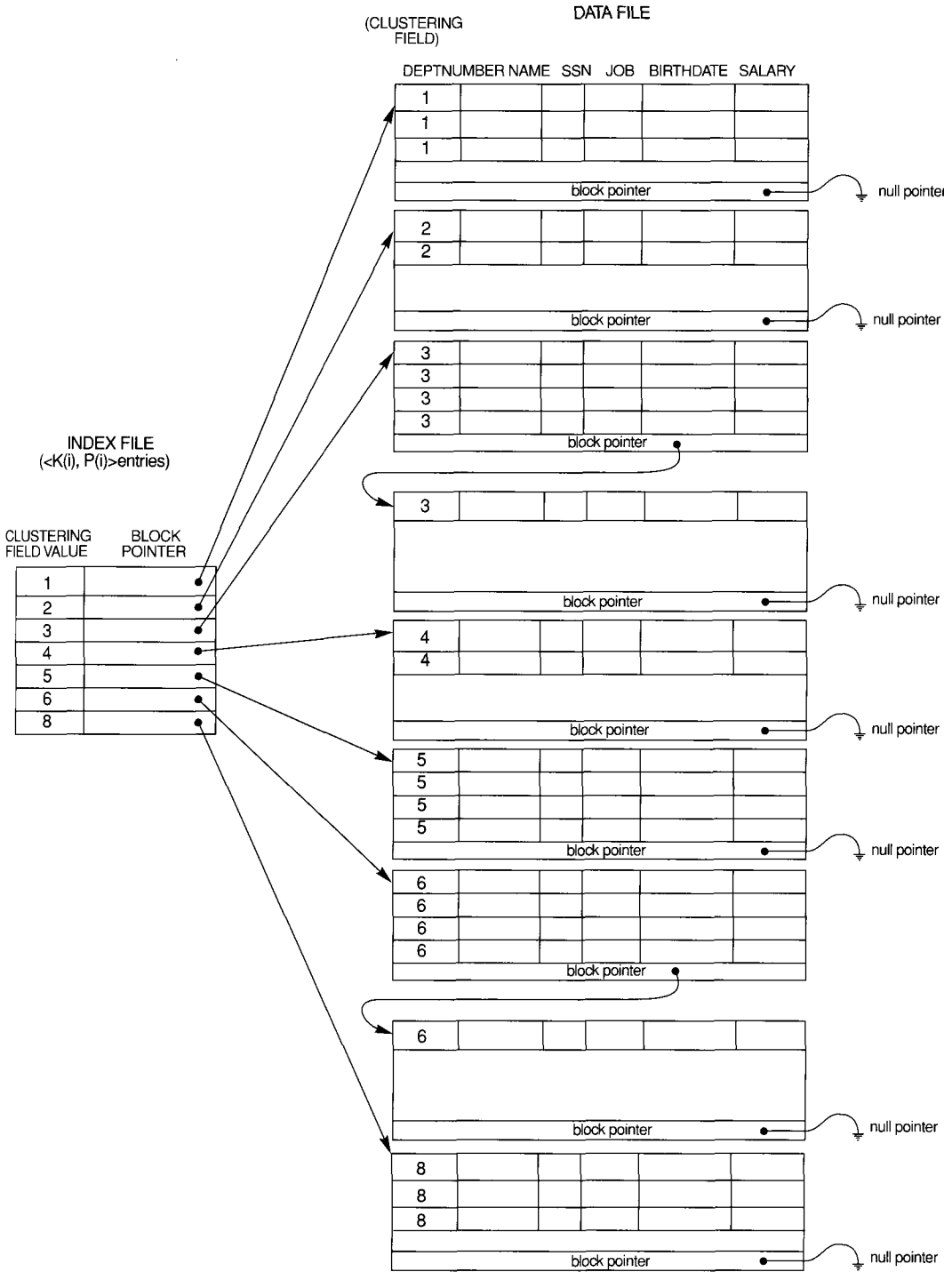


FIGURE 14.3 Clustering index with a separate block cluster for each group of records that share the same value for the clustering field.

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 hash value that is calculated by applying the hash function to the search field.

14.1.3 Secondary Indexes

A **secondary index** provides a secondary means of accessing a file for which some primary access already exists. The secondary index may be on a field which is a candidate key and has a unique value in every record, or a nonkey with duplicate values. The index is 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. There can be *many* secondary indexes (and hence, indexing fields) for the same file.

We first consider a secondary index access structure on a key field that has a *distinct value* for every record. Such a field is sometimes called a **secondary key.** In this case there is one index entry for *each record* in the data file, which contains the value of the secondary key 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.**

We again 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 14.4 illustrates a secondary index in which the pointers P(i) in the index entries are *block pointers*, not record pointers. Once the appropriate block is transferred to main memory, a search for the desired record within the block can be carried out.

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. To do a linear search on the file, we would require b/2 = 3000/2 = 1500 block accesses on the average. Suppose that we construct a secondary index on a nonordering key field of the file that is V = 9 bytes long. 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 $b_i = \lceil (r_i/bfr_i) \rceil = \lceil (30{,}000/68) \rceil = 442$ 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 seven block accesses required for the primary index.

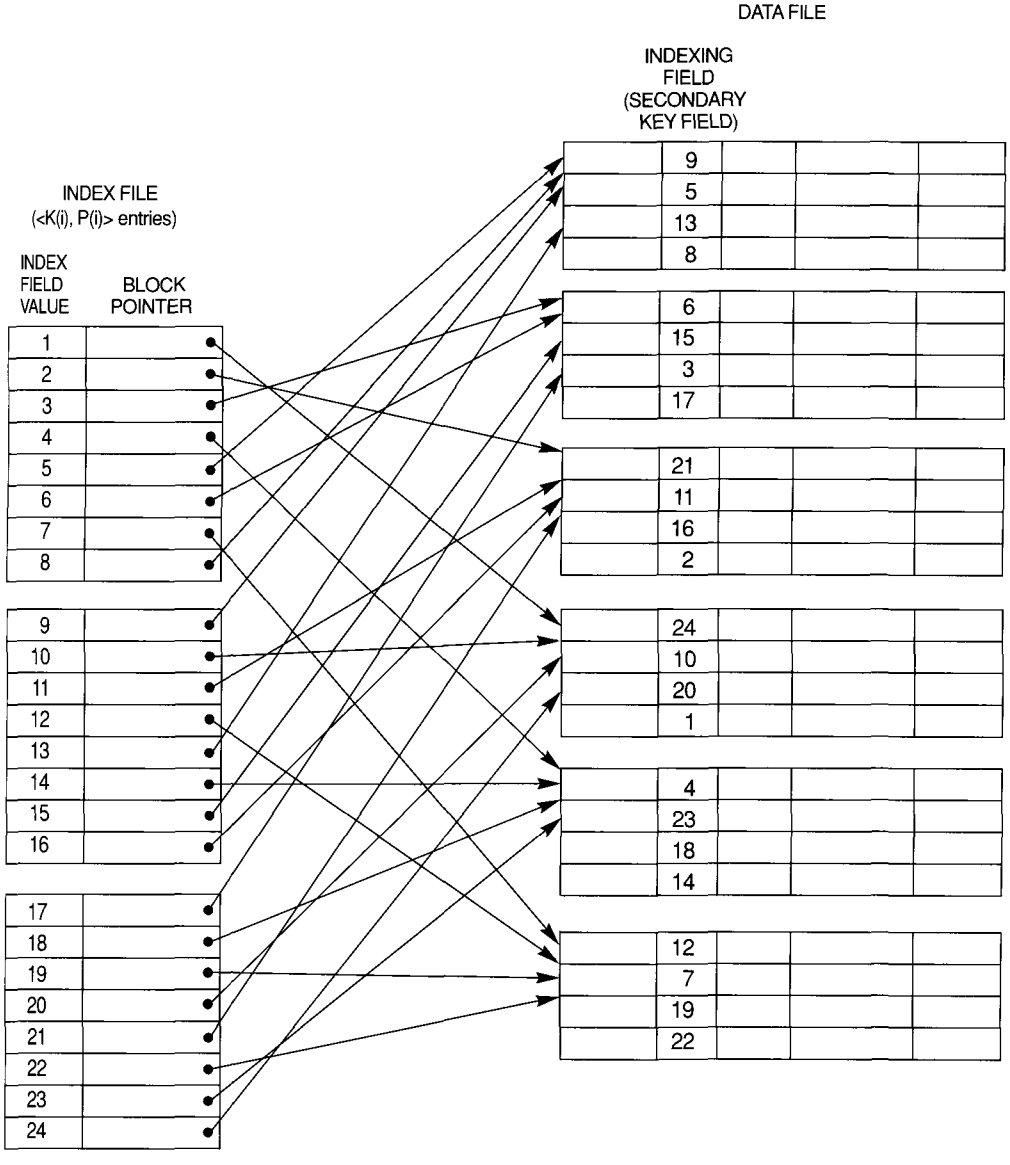


FIGURE 14.4 A dense secondary index (with block pointers) on a nonordering key field of a file.

We can also create a secondary index on a *nonkey 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 several 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), . . . , 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.
- 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 *block of record pointers*; each record pointer in that 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 14.5. 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 file records (see Exercise 14.19).

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.

14.1.4 Summary

To conclude this section, we summarize the discussion on index types in two tables. Table 14.1 shows the index field characteristics of each type of ordered single-level index discussed—primary, clustering, and secondary. Table 14.2 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.

14.2 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**. Searching a multilevel index requires approximately $(\log_{fo} b_i)$ block accesses, which is a smaller number than for binary search if the fan-out is larger than 2.

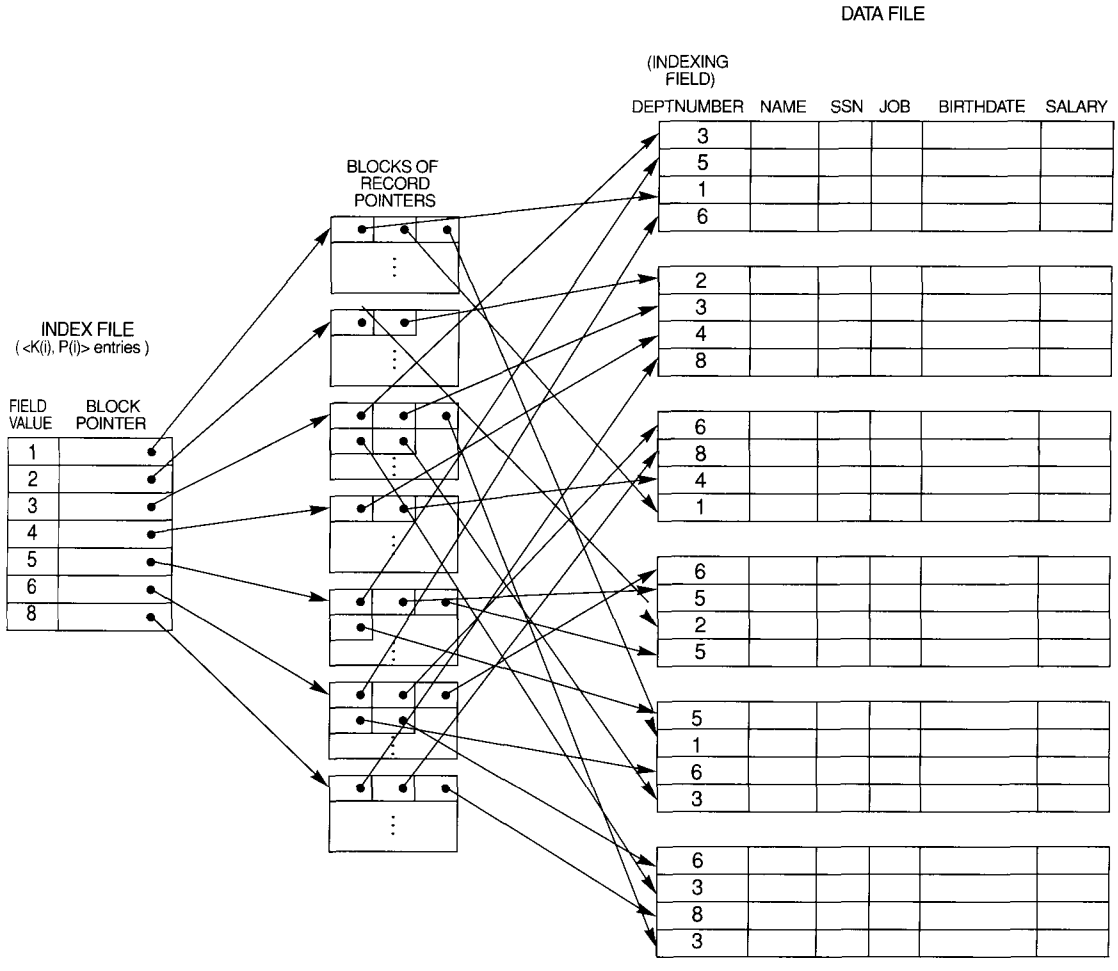


FIGURE 14.5 A secondary index (with record pointers) on a nonkey field implemented using one level of indirection so that index entries are of fixed length and have unique field values.

TABLE 14.1 TYPES OF INDEXES BASED ON THE PROPERTIES OF THE INDEXING FIELD

	INDEX FIELD USED FOR ORDERING THE FILE	INDEX FIELD NOT USED FOR ORDERING THE FILE
Indexing field is key	Primary index	Secondary index (Key)
Indexing field is nonkey	Clustering index	Secondary index (NonKey)

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). Hence we can create a primary index for the first level; this index to the first level is

TABLE 14.2 PROPERTIES OF INDEX TYPES

TYPE OF INDEX	NUMBER OF (FIRST-LEVEL) INDEX ENTRIES	DENSE OR NONDENSE	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.

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 t^{th} 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 $t = \lceil (\log_{fo}(r_1)) \rceil$.

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 14.6 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.

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, etc.

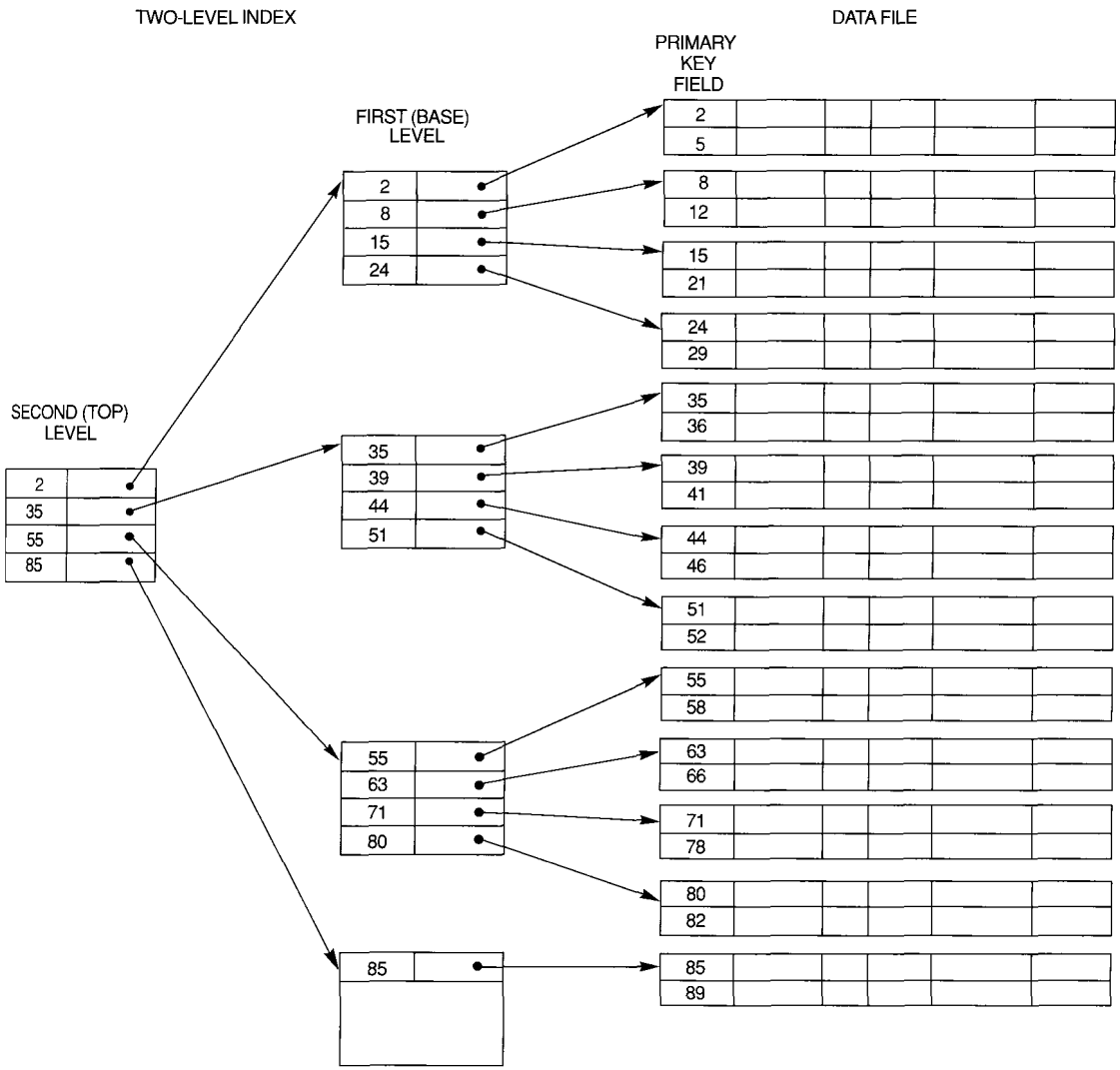


FIGURE 14.6 A two-level primary index resembling ISAM (Indexed Sequential Access Method) organization.

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 14.14(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. Insertion is handled by some form of overflow file that is merged periodically with the data file. The index is re-created during file reorganization. IBM's ISAM organization incorporates a two-level index that is closely related to the organization of the disk. The first level is a cylinder index, which has the key value of an anchor record for each cylinder of a disk pack 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.

Algorithm 14.1 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 14.19 discusses modifying the search algorithm for other types of indexes.

Algorithm 14.1: Searching a nondense multilevel primary index with t levels.

```
p ← address of top level block of index;
for j ← t step - 1 to 1 do
    begin
    read the index block (at j^th 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 j^th 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 that leaves some

space in each of its blocks for inserting new entries. This is called a **dynamic multilevel index** and is often implemented by using data structures called B-trees and B+-trees, which we describe in the next section.

14.3 Dynamic Multilevel Indexes Using B-Trees and B+-Trees

B-trees and B+-trees are special cases of the well-known tree data structure. We introduce very briefly 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 several—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 14.7 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.

Usually, we display a tree with the root node at the top, as shown in Figure 14.7. One way to implement a tree is to have as many pointers in each node as there are child nodes

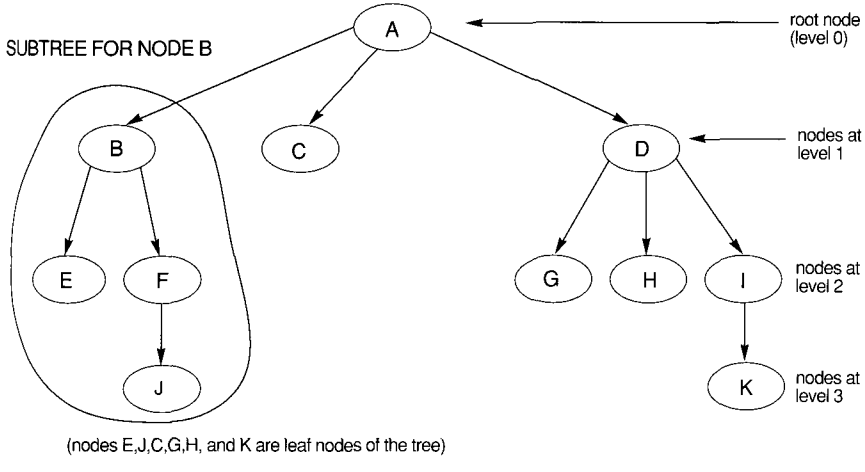


FIGURE 14.7 A tree data structure that shows an unbalanced tree.

5. This standard definition of the level of a tree node, which we use throughout Section 14.3, is different from the one we gave for multilevel indexes in Section 14.2.

of that node. In some cases, a parent pointer is also stored in each node. In addition to pointers, a node usually contains some kind of stored information. When a multilevel index is implemented as a tree structure, this information includes the values of the file's indexing field that are used to guide the search for a particular record.

In Section 14.3.1, 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 14.3.2 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; this can lead to fewer levels and higher-capacity indexes.

14.3.1 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 14.2 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 14.8 illustrates a node in a search tree. Two constraints must hold at all times on the search tree:

1. Within each node, $K_1 < K_2 < \ldots < K_{q-1}$.

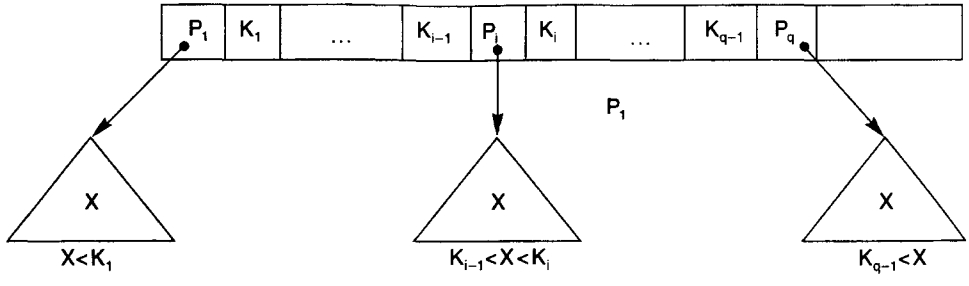


FIGURE 14.8 A node in a search tree with pointers to subtrees below it.

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.

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 14.8).

Whenever we search for a value X, we follow the appropriate pointer P_i according to the formulas in condition 2 above. Figure 14.9 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, 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.

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 14.7 is not balanced because it has leaf nodes at levels 1, 2, and 3. Keeping a search tree balanced is important because it guarantees that no nodes will be at very high levels and hence require many block accesses during a tree search. Keeping the tree balanced yields a uniform search speed regardless of the value of the search key. Another problem with search trees is that 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

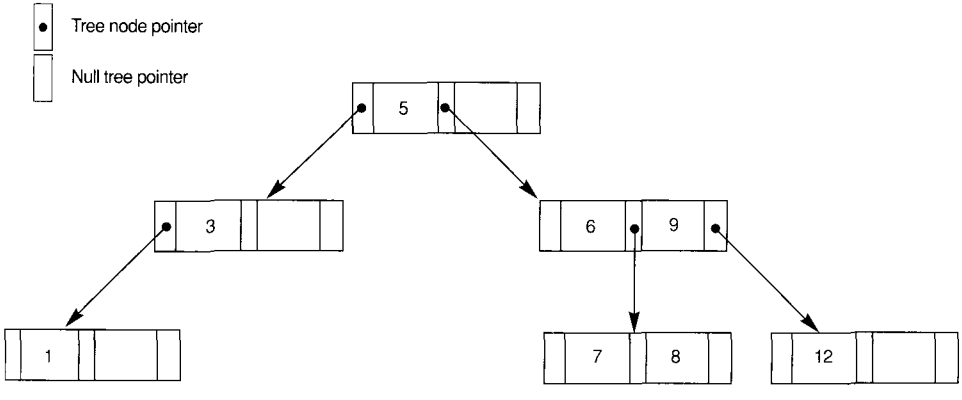


FIGURE 14.9 A search tree of order $p = 3$.

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

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 14.10a) is of the form

   $<P_1, <K_1, Pr_1>, P_2, <K_2, Pr_2>, \ldots, <K_{q-1}, Pr_{q-1}>, P_q>$

   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, $K_1 < K_2 < \ldots < K_{q-1}$.
3. For all search key field values X in the subtree pointed at by P_i (the i^{th} subtree, see Figure 14.10a), we have:

   $K_{i-1} < X < K_i$ for $1 < i < q$; $X < K_i$ for $i = 1$; and $K_{i-1} < X$ for $i = q$.
4. Each node has at most p tree pointers.
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 14.10b 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 cluster of blocks—that contain the pointers to the file records. This extra level of indirection is similar to Option 3, discussed in Section 14.1.3, 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 here; rather, we outline search and insertion procedures for B^+-trees in the next section.

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.

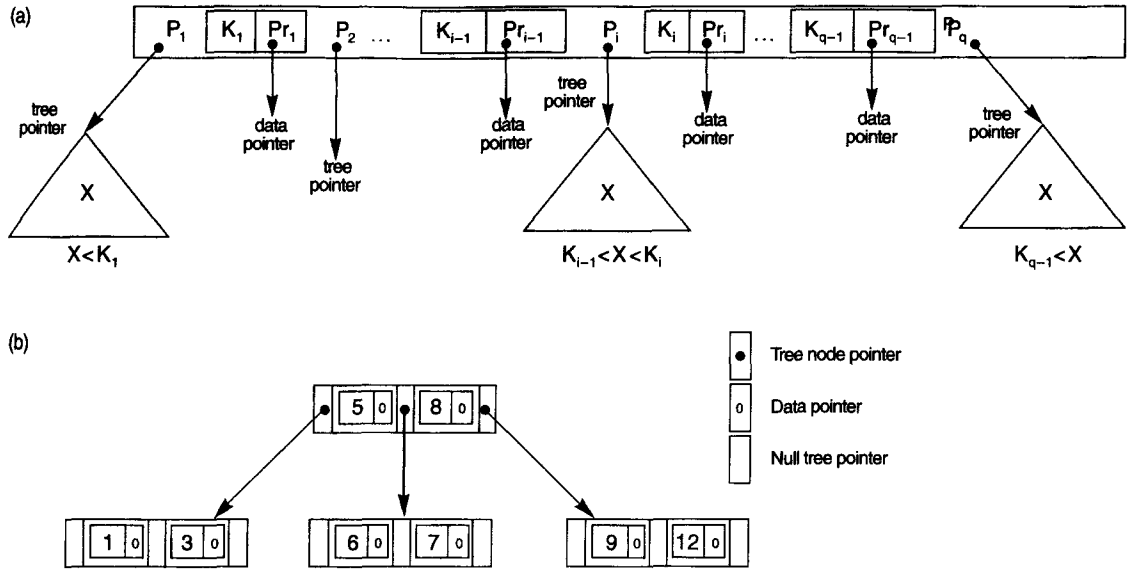


FIGURE 14.10 B-tree structures. (a) A node in a B-tree with q – 1 search values. (b) A B-tree of order p = 3. The values were inserted in the order 8, 5, 1, 7, 3, 12, 9, 6.

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. Example 4 illustrates how we calculate the order p of a B-tree stored on disk.

EXAMPLE 4: Suppose the search field is V = 9 bytes long, the disk block size is B = 512 bytes, a record (data) pointer is P_r = 7 bytes, and a block pointer is P = 6 bytes. Each B-tree node can have *at most* p tree pointers, p – 1 data pointers, and p – 1 search key field values (see Figure 14.10a). These must fit into a single disk block if each B-tree node is to correspond to a disk block. Hence, we must have:

$$(p * P) + ((p - 1) * (P_r + V)) \leq B$$

$$(p * 6) + ((p - 1) * (7 + 9)) \leq 512$$

$$(22 * p) \leq 528$$

We can choose p to be a large value that satisfies the above inequality, which gives p = 23 (p = 24 is not chosen because of the reasons given next).

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. Hence, before we do the preceding calculation for p, we should reduce the block size by the amount of space needed for all such information. Next, we illustrate how to calculate the number of blocks and levels for a B-tree.

EXAMPLE 5: Suppose that the search field of Example 4 is a nonordering key field, and we construct a B-tree on this field. 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 entries	16 pointers
Level 1:	16 nodes	240 entries	256 pointers
Level 2:	256 nodes	3840 entries	4096 pointers
Level 3:	4096 nodes	61,440 entries	

At each level, we calculated the number of 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 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.

14.3.2 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 together 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 14.11a) is as follows:

1. Each internal node is of the form

   $<P_1, K_1, P_2, K_2, \ldots, P_{q-1}, K_{q-1}, P_q>$

   where $q \leq p$ and each P_i is a **tree pointer**.
2. Within each internal node, $K_1 < K_2 < \ldots < K_{q-1}$.
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 14.11a).[9]
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 14.11b) is as follows:

1. Each leaf node is of the form

   $<<K_1, Pr_1>, <K_2, Pr_2>, \ldots, <K_{q-1}, Pr_{q-1}>, P_{next}>$

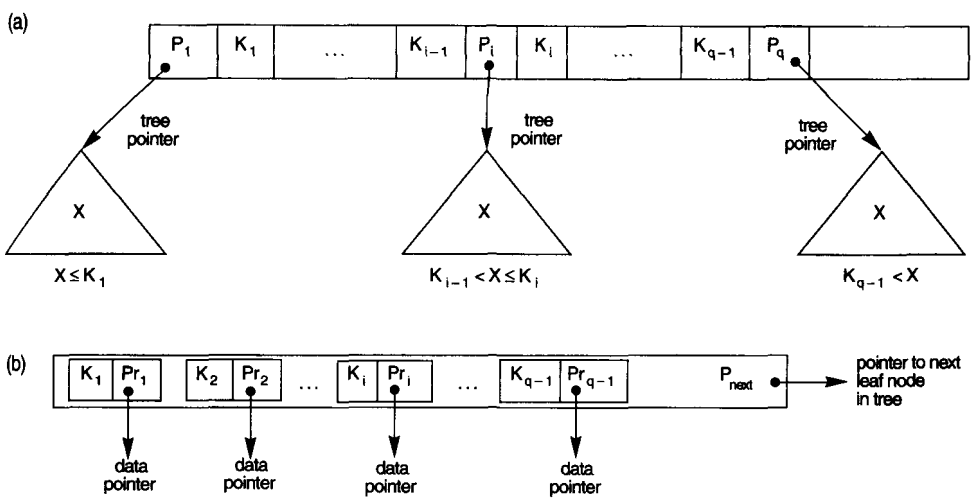


FIGURE 14.11 The nodes of a B+-tree. (a) Internal node of a B+-tree with $q - 1$ search values. (b) Leaf node of a B+-tree with $q-1$ search values and $q-1$ data pointers.

9. Our definition follows Knuth (1973). One can define a B+-tree differently by exchanging the $<$ and $\leq$ symbols ($K_{i-1} \leq X < K_i$; $X < K_1$; $K_{q-1} \leq X$), but the principles remain the same.

where $q \leq p$, each Pr_i is a data pointer, and P_{next} points to the next *leaf node* of the B^+-tree.

2. Within each leaf node, $K_1 < K_2 < ... < K_{q-1}$, $q \leq p$.
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 14.5, 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 14.1.3.

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 6. 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 6: 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 $P_r = 7$ bytes, and a block pointer is $P = 6$ bytes, as in Example 4. 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, 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:

$$(p_{leaf} * (P_r + V)) + P \leq B$$

$(p_{leaf} * (7 + 9)) + 6 \leq 512$

$(16 * p_{leaf}) \leq 506$

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 7: Suppose that we construct a B+-tree on the field of Example 6. To calculate the approximate number of entries of 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 entries	23 pointers
Level 1:	23 nodes	506 entries	529 pointers
Level 2:	529 nodes	11,638 entries	12,167 pointers
Leaf level:	12,167 nodes	255,507 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, on the average. Compare this to the 65,535 entries for the corresponding B-tree in Example 5.

Search, Insertion, and Deletion with B+-Trees.

Algorithm 14.2 outlines the procedure using the B+-tree as access structure to search for a record. Algorithm 14.3 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 now illustrate insertion and deletion with an example.

Algorithm 14.2: 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.K_{i-1} < K # n.K_i;
        n ← n.P_i
        end;
    read block n
    end;
search block n for entry (K_i,Pr_i) with K = K_i; (* search leaf node *)
if found
    then read data file block with address Pr_i and retrieve record
      else record with search field value K is not in the data file;
```

Algorithm 14.3: 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.K_1 (*n.K_i refers to the i^th search field value in node n*)
     then n ← n.P_1 (*n.P_i refers to the i^th tree pointer in node n*)
     else if K > n.K_{q-1}
         then n ← n.P_q
         else begin
         search node n for an entry i such that n.K_{i-1} < K # n.K_i;
         n ← n.P_i
         end;
      read block n
    end;
search block n for entry (K_i,Pr_i) with K = K_i; (*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 p_leaf record pointers-is split*)
    copy n to temp (*temp is an oversize leaf node to hold extra
entry*);
    insert entry (K, Pr) in temp in correct position;
      (*temp now holds p_leaf + 1 entries of the form (K_i, Pr_i)*)
    new ← a new empty leaf node for the tree; new.P_next ← n.P_next;
    j ← ⌈(p_leaf + 1)/2⌉ ;
    n ← first j entries in temp (up to entry (K_j,Pr_j)); n.P_next ← new;
    new ← remaining entries in temp; K ← K_j;
```

```
(*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-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 Pj in temp;
             (*n contains <P1, K1, P2, K2, ..., Pj-1, Kj-1, Pj >*)
       new ← entries from tree pointer Pj+1 in temp;
          (*new contains < Pj+1, Kj+1, ..., Kp-1, Pp, Kp, Pp+1 >*)
       K ← Kj
         (*now we must move (K,new) and insert in parent internal node*)
       end
    end
   until finished
     end;
end;
```

Figure 14.12 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,

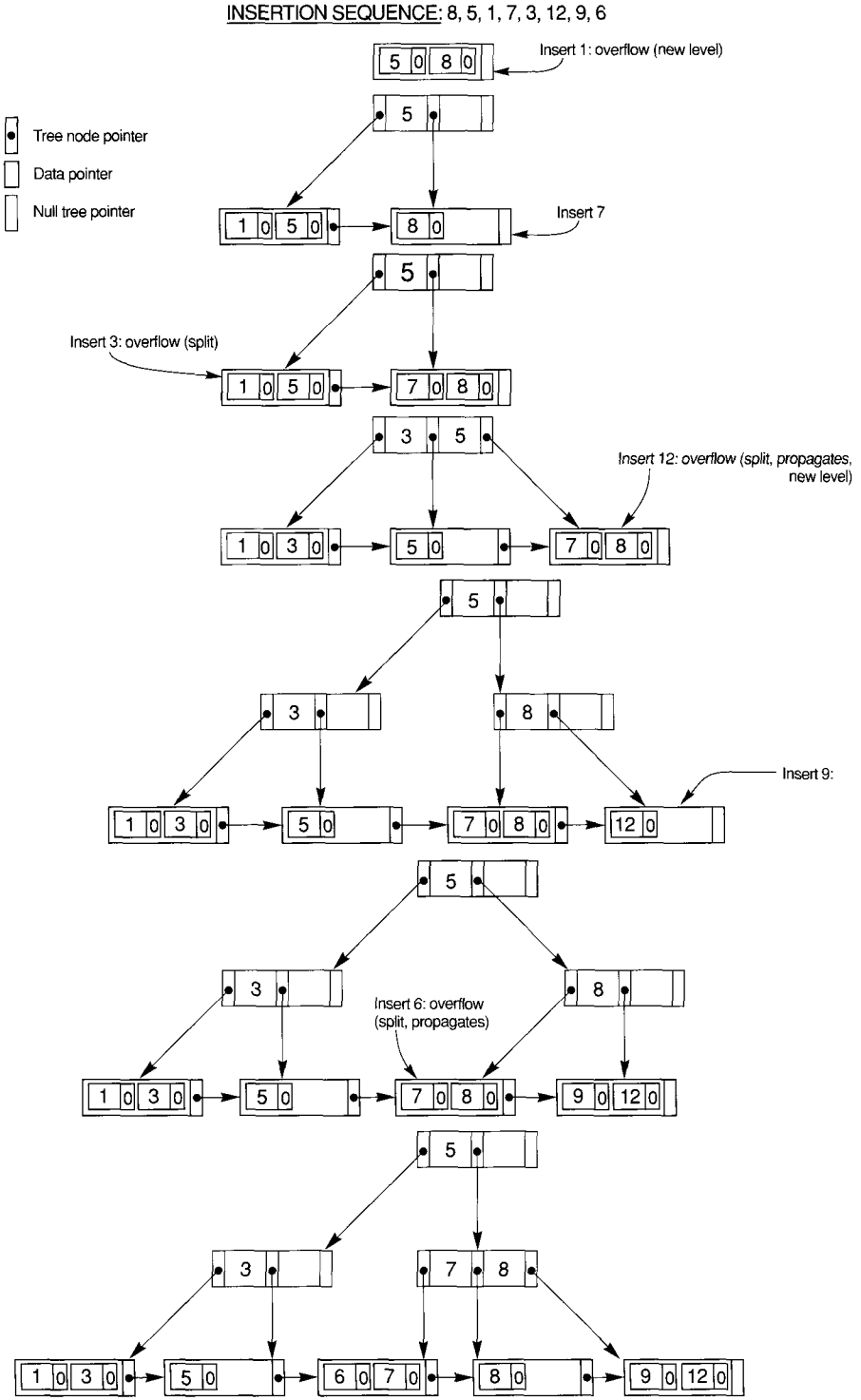


FIGURE 14.12 An example of insertion in a B^+-tree with $p = 3$ and $p_{leaf} = 2$.

and the remaining entries are moved to a new leaf node. The j^{th} 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 j^{th} tree pointer after inserting the new value and pointer, where $j = \lfloor ((p + 1)/2) \rfloor$—are kept, while the j^{th} 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 14.3). This splitting can propagate all the way up to create a new root node and hence a new level for the B+-tree.

Figure 14.13 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—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 redistributing entries with the left sibling; if this is not possible, an attempt to redistribute with the right sibling is made. If this is not possible either, 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 14.3. 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.

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 B+-tree), 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. Recently, 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.

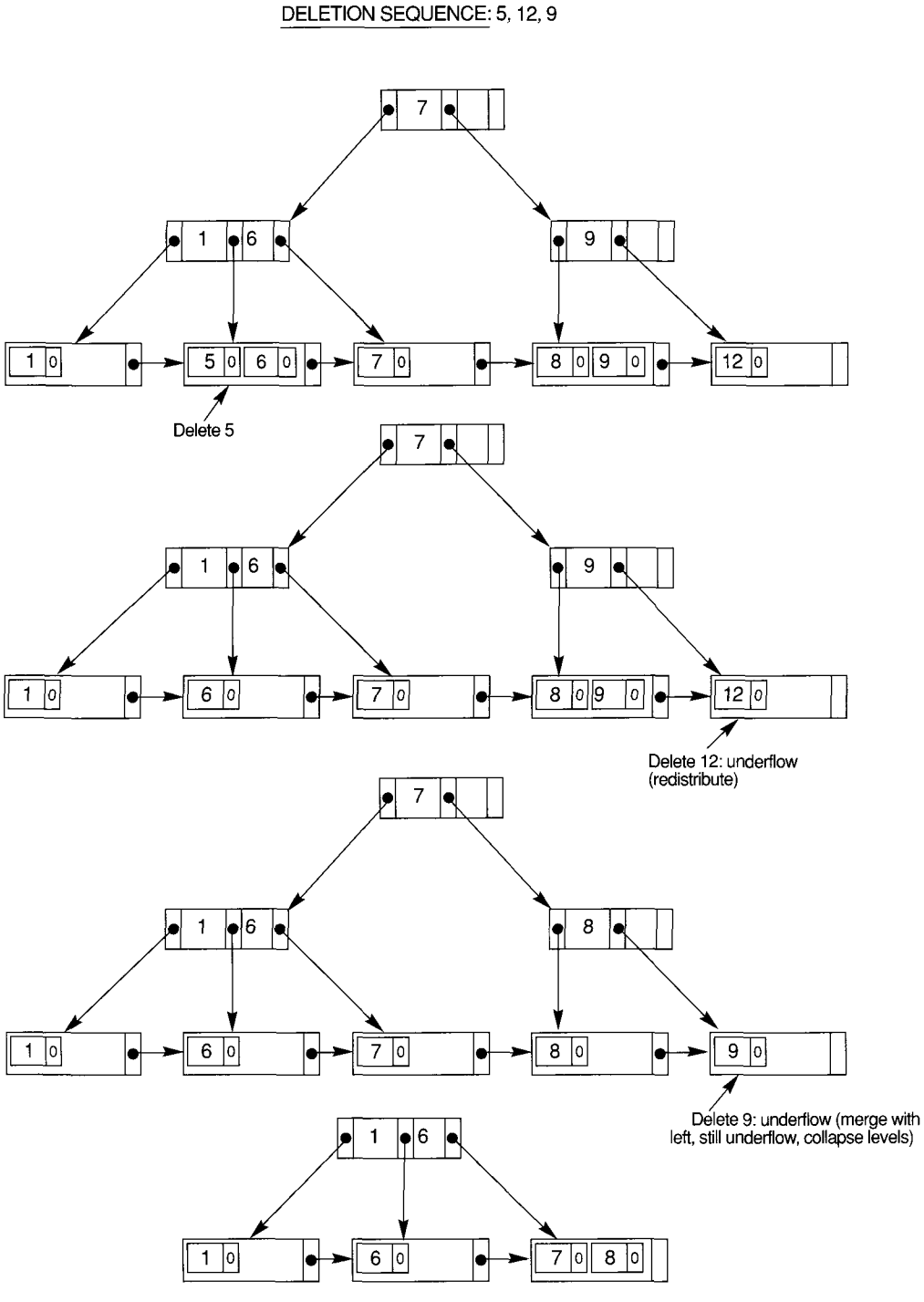


FIGURE 14.13 An example of deletion from a B^+-tree.

14.4 INDEXES ON MULTIPLE KEYS

In our discussion so far, we 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 very 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, ZIPCODE, 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 then select from among them those records that satisfy AGE = 59.
2. Alternately, if AGE is indexed but DNO is not, access the records having AGE = 59 using the index 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 that satisfy both conditions, those records that satisfy both conditions, or the blocks in which records satisfying both conditions are located.

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 a very 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 below. We will refer to keys containing multiple attributes as **composite keys**.

14.4.1 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: $<v_1, v_2, \ldots\ldots, v_n>$.

A lexicographic ordering of these tuple values establishes an order on this composite search key. For our example, all of department keys for department number 3 precede those for department 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.

14.4.2 Partitioned Hashing

Partitioned hashing is an extension of static external hashing (Section 13.8.2) 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", . . . etc. 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.

14.4.3 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 14.14 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

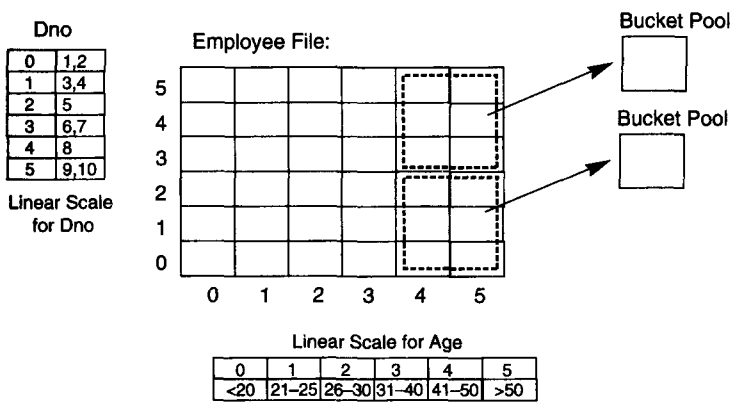


FIGURE 14.14 Example of a grid array on DNO and AGE attributes.

bucket address where the records corresponding to that cell are stored. Figure 14.14 also shows 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. Conceptually, the grid file concept may be applied to any number of search keys. 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.[10]

14.5 Other Types of Indexes

14.5.1 Using Hashing and Other Data Structures as Indexes

It is also possible to create access structures similar to indexes that are based on *hashing*. The index entries <K, Pr> (or <K, P>) can be organized as a dynamically expandable hash file, using one of the techniques described in Section 13.8.3; 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. Other search structures can also be used as indexes.

14.5.2 Logical versus Physical Indexes

So far, 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—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

10. Insertion/deletion algorithms for grid files may be found in Nievergelt [1984].

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.

14.5.3 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.

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 ADABAS of Software-AG, use this method extensively.

We referred to the popular IBM file organization called ISAM in Section 14.2. Another IBM method, the **virtual storage access method (VSAM)**, is somewhat similar to the B^+-tree access structure.

14.6 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 13, which are used to organize the file records themselves on disk.

Three types of ordered single-level indexes were introduced: (1) primary, (2) clustering, and (3) 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. 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.

We then 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 then introduced the concept of a logical index, and compared it with the physical indexes we described before. 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. Secondary indexes can also be created for hash files and dynamic hash files.

Review Questions

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

14.2. 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?

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

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

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

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

14.7. 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?

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

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

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

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

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

14.13. How can hashing be used to construct an index? What is the difference between a logical index and a physical index?

Exercises

14.14. 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), DEPARTMENTCODE (9 bytes), ADDRESS (40 bytes), PHONE (9 bytes), BIRTHDATE (8 bytes), SEX (1 byte), JOBCODE (4 bytes), 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 DEPARTMENTCODE and we want to construct a *secondary index* on DEPARTMENTCODE, using option 3 of Section 14.1.3, with an extra level of indirection that stores record pointers. Assume there are 1000 distinct values of DEPARTMENTCODE 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 DEPARTMENTCODE value, using the index.

f. Suppose that the file is *ordered* by the nonkey field DEPARTMENTCODE and we want to construct a *clustering index* on DEPARTMENTCODE that uses block anchors (every new value of DEPARTMENTCODE starts at the beginning of a new block). Assume there are 1000 distinct values of DEPARTMENTCODE 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 DEPARTMENTCODE 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.

14.15. A PARTS file with Part# as 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.

14.16. Repeat Exercise 14.15, but use a B-tree of order p = 4 instead of a B^+-tree.

14.17. Suppose that the following search field values are deleted, in the given order, from the B^+-tree of Exercise 14.15; show how the tree will shrink and show the final tree. The deleted values are 65, 75, 43, 18, 20, 92, 59, 37.

14.18. Repeat Exercise 14.17, but for the B-tree of Exercise 14.16.

14.19. Algorithm 14.1 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 14.1.3 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.

14.20. Suppose that several secondary indexes exist on nonkey fields of a file, implemented using option 3 of Section 14.1.3; for example, we could have secondary indexes on the fields DEPARTMENTCODE, JOBCODE, and SALARY of the EMPLOYEE file of Exercise 14.14. Describe an efficient way to search for and retrieve records satisfying a complex selection condition on these fields, such as (DEPARTMENTCODE = 5 AND JOBCODE = 12 AND SALARY = 50,000), using the record pointers in the indirection level.

14.21. Adapt Algorithms 14.2 and 14.3, which outline search and insertion procedures for a B^+-tree, to a B-tree.

14.22. 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 14.15 illustrates how this could be done for our example in Figure 14.12; 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 14.15 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.

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

14.24. Repeat Exercise 14.23 for a B-tree.

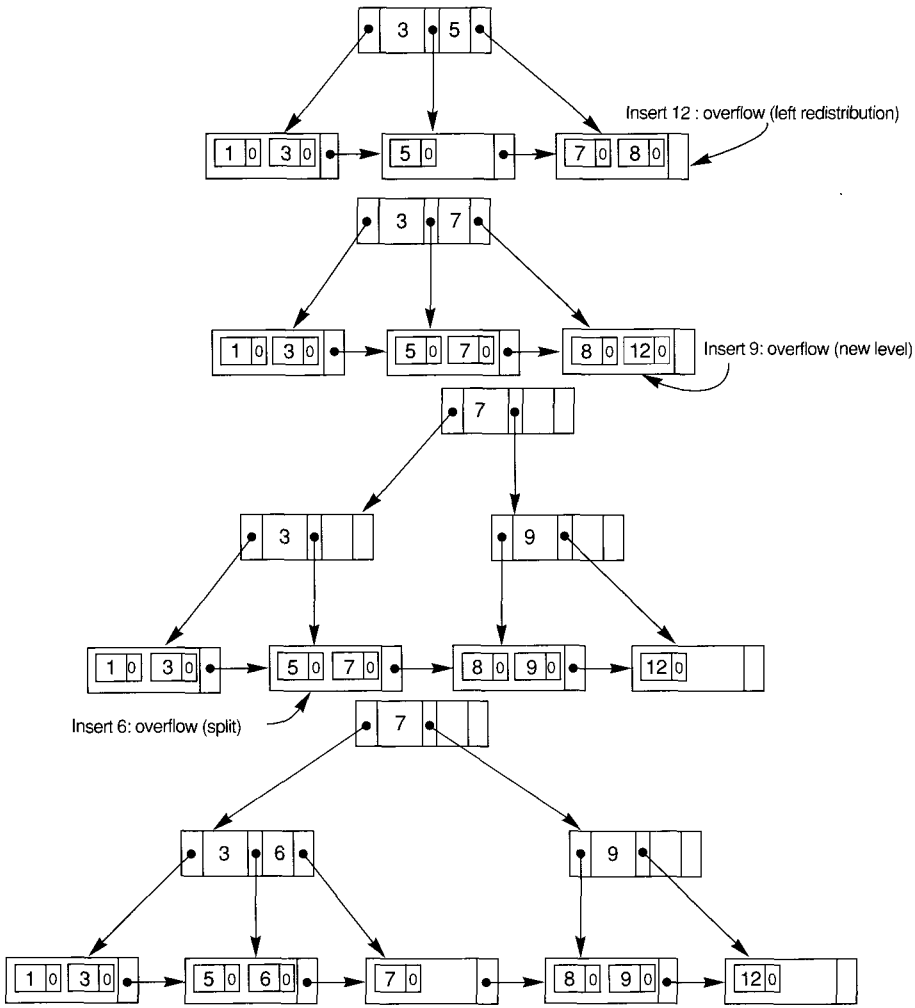


FIGURE 14.15 B^+-tree insertion with left redistribution.

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 (1973) 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 Wirth (1972), Claybrook (1983), Smith and Barnes (1987), Miller (1987), and Salzberg (1988) 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 (1983), 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 (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).

15

Algorithms for Query Processing and Optimization

In this chapter we discuss the techniques used 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 language tokens—such as SQL keywords, attribute names, and relation names—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** for retrieving the result 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 15.1 shows the different steps of processing a high-level query. The **query optimizer** module has the task of producing an execution plan, and the **code generator** generates the code to execute that plan. The **runtime database processor** has the task of running the query code,

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

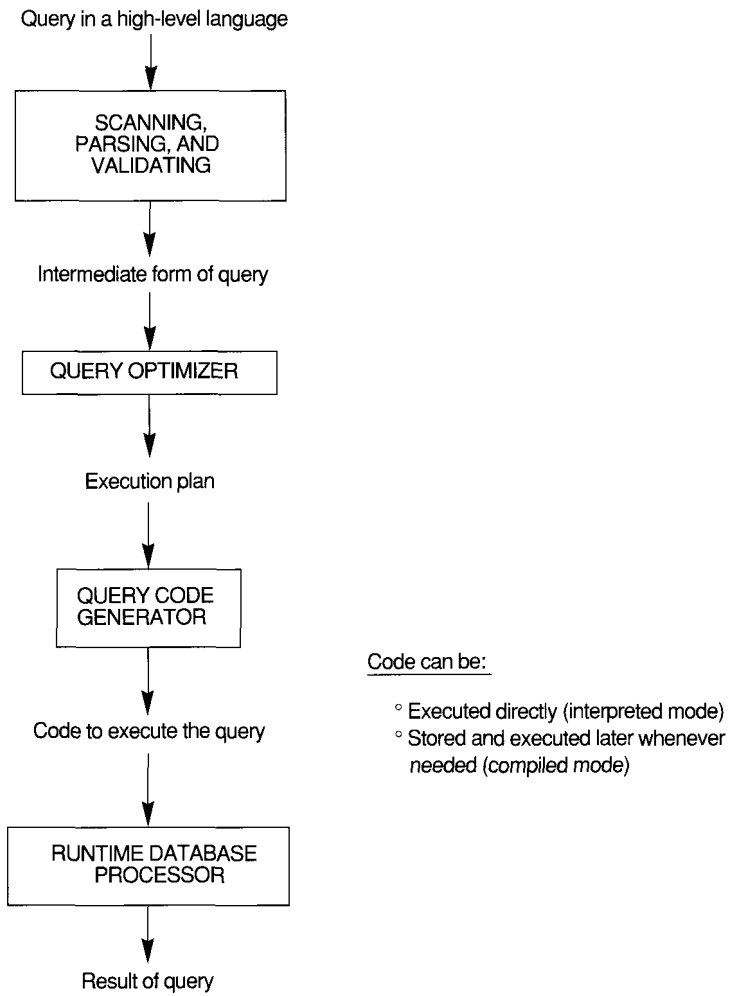


FIGURE 15.1 Typical steps when processing a high-level query.

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.

The term *optimization* is actually a misnomer because in some cases the chosen execution plan is not the optimal (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 and may require 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 an 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 HDML (see Appendixes E and F)—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 "optimal" execution strategy. On the other hand, a high-level query language—such as SQL for relational DBMSs (RDBMSs) or OQL (see Chapter 21) 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 been adapted for ODBMSs.[2] A relational DBMS must systematically evaluate alternative query execution strategies and choose a reasonably efficient or optimal strategy. Each DBMS typically has a number of general database access algorithms that implement relational operations such as SELECT or JOIN or combinations of these operations. Only execution strategies that can be implemented by the DBMS access algorithms and that apply to the particular query and particular physical database design can be considered by the query optimization module.

We start in Section 15.1 with a general discussion of how SQL queries are typically translated into relational algebra queries and then optimized. We then discuss algorithms for implementing relational operations in Sections 15.2 through 15.6. Following this, we give an overview of query optimization strategies. There are two main techniques for implementing 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 possible 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. The two techniques are usually combined in a query optimizer. We discuss heuristic optimization in Section 15.7 and cost estimation in Section 15.8. We then provide a brief overview of the factors considered during query optimization in the ORACLE commercial RDBMS in Section 15.9. Section 15.10 introduces the topic of semantic query optimization, in which known constraints are used to devise efficient query execution strategies.

15.1 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

2. There are some query optimization problems and techniques that are pertinent only to ODBMSs. However, we do not discuss these here as we can give only an introduction to query optimization.

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 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 5.5:

```
SELECT  LNAME, FNAME
FROM    EMPLOYEE
WHERE   SALARY >  (SELECT    MAX (SALARY)
                   FROM      EMPLOYEE
                   WHERE     DNO=5);
```

This 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)
```

and the outer 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 extended relational algebra expression

$$\mathfrak{F}_{\text{MAX SALARY}}(\sigma_{\text{DNO}=5}(\text{EMPLOYEE}))$$

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 block. We should note that in the above example, the inner block needs to be evaluated only once to produce the maximum salary, which is then used—as the constant c—by the outer block. We called this an *uncorrelated nested query* in Chapter 8. It is much harder to optimize the more complex *correlated nested queries* (see Section 8.5), where a tuple variable from the outer block appears in the WHERE-clause of the inner block.

15.2 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 may be avoided if an appropriate index exists to allow ordered access to the records.

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 15.2, consists of two phases: (1) the sorting phase and (2) the merging phase.

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 a run and **number of initial runs (n_R)** is dictated by the **number of file blocks (b)** and the **available buffer**

```
set   i ← 1;
      j ← b;    {size of the file in blocks}
      k ← nB;   {size of buffer in blocks}
      m ← ⌈(j/k)⌉;
{Sort 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;
         }
{Merge 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 15.2 Outline of the sort-merge algorithm for external sorting.

3. *Internal sorting algorithms* are suitable for sorting data structures that can fit entirely in memory.

space (n_B). For example, if n_B = 5 blocks and the size of the file b = 1024 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 4 blocks). Hence, after the sort phase, 205 sorted runs are stored as temporary subfiles on disk.

In the **merging phase,** the sorted runs are merged during one or more **passes.** The **degree of merging (d_M)** is the number of runs that can be merged together in each pass. In each pass, one buffer block is needed to hold one block from each of the runs being merged, and one block is needed for containing one block of the merge result. Hence, d_M is the smaller of (n_B − 1) and n_R, and the number of passes is $\lceil (log_{dM}(n_R)) \rceil$. In our example, d_M = 4 (four-way merging), so the 205 initial sorted runs would be merged into 52 at the end of the first pass, which are then merged into 13, then 4, then 1 run, which means that *four passes* are needed. The minimum d_M of 2 gives the worst-case performance of the algorithm, which is

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

The first term represents the number of block accesses for the sort phase, since each file block is accessed twice—once for reading into memory and once for writing the records back to disk after sorting. The second term represents the number of block accesses for the merge phase, assuming the worst-case d_M of 2. In general, the log is taken to the base d_M and the expression for number of block accesses becomes

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

15.3 Algorithms for SELECT and JOIN Operations

15.3.1 Implementing the SELECT Operation

There are many options for executing a SELECT operation; some depend on the file having specific access paths and 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 of Figure 5.5, to illustrate our discussion:

(OP1): $\sigma_{SSN='123456789'}$(EMPLOYEE)
(OP2): $\sigma_{DNUMBER>5}$(DEPARTMENT)
(OP3): $\sigma_{DNO=5}$(EMPLOYEE)
(OP4): $\sigma_{DNO=5\ AND\ SALARY>30000\ AND\ SEX='F'}$(EMPLOYEE)
(OP5): $\sigma_{ESSN='123456789'\ AND\ PNO=10}$(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]

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.

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):* Retrieve *every record* in the file, and test whether its attribute values 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]
- S3. *Using a primary index (or hash key):* If the selection condition involves an equality comparison on a **key attribute** with a primary index (or hash key)—for example, SSN = '123456789' in OP1—use the primary index (or 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 **non-key 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 15.8, we discuss how to develop formulas that estimate the access cost of these search methods in terms of number of block accesses and access time. Method S1 applies to any file, but all the other methods depend on having the appropriate access path on the attribute used in the selection condition. 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 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 condition has an access path that

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

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 condition.

- S8. *Conjunctive selection using a composite index:* If two or more attributes are involved in equality conditions in the conjunctive 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 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 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]

Whenever a single condition specifies the selection—such as OP1, OP2, or OP3—we can only check whether an access path exists on the attribute involved in that condition. If an access path 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 15.8) and choosing the method with the least estimated cost.

When the optimizer is choosing between multiple simple conditions in a conjunctive select condition, it typically considers the selectivity of each condition. The **selectivity (s)** 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 1—zero selectivity means no records satisfy the condition and 1 means all the records satisfy the condition. 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 an attribute with i *distinct values*, s can be estimated by $(|r(R)|/i)/|r(R)|$ or

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.

$1/i$, assuming that the records are evenly 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 s is estimated to be $|r(R)| * s$. The smaller this estimate is, the higher the desirability of using that condition first to retrieve records.

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′:

(OP49): $\sigma_{\text{DNO=5 OR SALARY>30000 OR SEX='F'}}(\text{EMPLOYEE})$

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* condition 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 shall discuss in Section 15.8. The optimizer chooses the access method with the lowest estimated cost.

15.3.2 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 only these two here. 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 of Figure 5.5 once more—specifically, to the EMPLOYEE, DEPARTMENT, and PROJECT relations. The algorithms we consider are for join operations of the form

$R \bowtie_{A=B} S$

where A and B are 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 example operations:

(OP6): $\text{EMPLOYEE} \bowtie_{\text{DNO=DNUMBER}} \text{DEPARTMENT}$

(OP7): $\text{DEPARTMENT} \bowtie_{\text{MGRSSN=SSN}} \text{EMPLOYEE}$

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

Methods for Implementing Joins

- J1. *Nested-loop join (brute force):* 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, B of S—retrieve each record t in R, one at a time (single loop), and then use the access structure 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 15.2). 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 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 15.3a. We use $R(i)$ to refer to the i^{th} record in 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. *Hash-join:* The records of files R and S are both hashed to the same hash file, using the same hashing function on the join attributes A of R and B of S as hash keys. First, a single pass through the file with fewer records (say, R) hashes its records to the hash file buckets; this is called the **partitioning phase,** since the records of R are partitioned into the hash buckets. In the second phase, called the **probing phase,** a single pass through the other file (S) then hashes each of its records to *probe* the appropriate bucket, and that record is combined with all matching records from R in that bucket. This simplified description of hash-join assumes that the smaller of the two files *fits entirely into memory buckets* after the first phase. We will discuss variations of hash-join that do 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 buffer space in memory, the number of blocks read in from the file can be adjusted.

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*.

```
(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← i+1, j←j+1
                   }
     }

(b) create a tuple t[<attribute list>] in T' for each tuple t in R;
        (*T' contains the projection result 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 *)
```

FIGURE 15.3 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 $T \leftarrow \pi_{<\text{attribute list}>}(R)$.

Effects of Available Buffer Space and Join Selection Factor on Join Performance. The buffer space available has an important effect on the various 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). 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

```
(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(i) 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;
```

FIGURE 15.3(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 $T \leftarrow R - S$.

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 in memory for matching records. This reduces the total number of block accesses. An extra buffer block is needed to contain the resulting records after they are joined, and the con-

tents of this buffer block are appended to the **result file**—the disk file that contains the join result—whenever it is filled. This buffer block is then reused to hold additional result records.

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:

Total number of blocks accessed for outer file = b_E

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

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

Hence, we get the following total number of block 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 reused.[10] If the result file of the join operation has b_{RES} disk blocks, each block is written once, so an additional b_{RES} block accesses 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.

Another factor that affects the performance of a join, particularly the single-loop method J2, is the percentage of records in a 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) is expected to be joined with a *single* EMPLOYEE record, but many EMPLOYEE records (the 5950 of them that do not manage a department) will not be joined.

Suppose that secondary indexes exist on both the attributes SSN of EMPLOYEE and MGRSSN of DEPARTMENT, with the number of index levels $x_{SSN} = 4$ and $x_{MGRSSN} = 2$, respectively. We have two options for implementing method J2. The first retrieves each EMPLOYEE record and then uses the index on MGRSSN of DEPARTMENT to find a matching DEPARTMENT record. In this case, no

10. If we reserve two buffers for the result file, double buffering can be used to speed the algorithm (see Section 13.3).

11. This is different from the *join selectivity*, which we shall discuss in Section 15.8.

matching record will be found for employees who do not manage a department. The number of block accesses for this case is approximately

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

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 = MGRSSN is 1, whereas the join selection factor of EMPLOYEE with respect to the same join condition is (50/6000), or 0.008. 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 (outer) 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, they may be sorted specifically for performing the join operation. If we 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]

Partition Hash Join and Hybrid 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, parts of the hash file 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: partition hash join and a variation called hybrid hash join, which has been shown to be quite efficient.

In the **partition hash join** algorithm, each file is first partitioned into M partitions using a **partitioning hash function** on the join attributes. Then, each pair of 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 $R_1, R_2, \ldots, R_M$, and S into the M partitions $S_1, S_2, \ldots, S_M$. The property of each pair of corresponding partitions R_i, S_i 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

12. We can use the more accurate formulas from Section 15.2 if we know the number of available buffers for sorting.

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. For each of the partitions, a single in-memory buffer—whose size is one disk block—is allocated to store the records that hash to this partition. Whenever the in-memory buffer for a partition gets filled, its contents are appended to a **disk subfile** that stores this partition. The partitioning phase has *two iterations*. After the first iteration, the first file R is partitioned into the subfiles $R_1, R_2, \ldots, R_M$, 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, the two 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. 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.

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

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 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. Hence, at the end of the second pass of the partitioning phase, all records that hash to the first partition have been joined. Now 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 back to disk and rereading them a second time during the joining phase.

15.4 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 is usually 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 15.3b. 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 already in the bucket; if it is a duplicate, it is not inserted. It is useful to recall here that in SQL queries, the default is not to eliminate duplicates from the query result; only if the keyword DISTINCT is included are duplicates eliminated from the query result.

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. In addition, the attributes of the result include 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 will have $n * m$ records and $j + k$ attributes. Hence, it is important to avoid the CARTESIAN PRODUCT operation and to substitute other equivalent operations during query optimization (see Section 15.7).

The other three set operations—UNION, INTERSECTION, and SET DIFFERENCE[14]—apply only to 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 relations*. Figure 15.3c 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 partitioned and the other is 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.

15.5 Implementing Aggregate Operations and Outer Joins

15.5.1 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) index on SALARY exists for the EMPLOYEE relation, then the optimizer can decide on using the index to search for the largest value by following the *rightmost* pointer in each index node from the root to the rightmost leaf. That node would include

14. SET DIFFERENCE is called EXCEPT in SQL.

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 aggregate can be handled in a similar manner, except that the *leftmost* pointer 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 COUNT, AVERAGE, and SUM aggregates, 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 entry must be used for a correct computation (except for COUNT DISTINCT, where the number of distinct values can be counted from the index itself).

When a GROUP BY clause is used in a query, the aggregate operator must be applied separately to each group of tuples. 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 example query, the set of 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 13) 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.

15.5.2 Implementing Outer Join

In Section 6.4, the *outer join operation* was introduced, with its three variations: left outer join, right outer join, and full outer join. We also discussed in Chapter 8 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.

Alternatively, outer join can 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.

   TEMP1 $\leftarrow \pi_{\text{LNAME, FNAME, DNAME}}$ (EMPLOYEE $\bowtie_{\text{DNO=DNUMBER}}$ DEPARTMENT)

2. Find the EMPLOYEE tuples that do not appear in the (inner) JOIN result.

   TEMP2 $\leftarrow \pi_{\text{LNAME, FNAME}}$ (EMPLOYEE) $-$ $\pi_{\text{LNAME, FNAME}}$ (TEMP1)

3. Pad each tuple in TEMP2 with a null DNAME field.

   TEMP2 $\leftarrow$ TEMP2 $\times$ 'NULL'

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

   RESULT $\leftarrow$ TEMP1 $\cup$ TEMP2

The cost of the outer join as computed above would be the sum of the costs of the associated steps (inner join, projections, 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.

15.6 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 correspond 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 15.7.2 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.

15.7 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 parser of a high-level query first generates an *initial internal representation*, which is then optimized according to heuristic rules. 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. This is 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.

We start in Section 15.7.1 by introducing the query tree and query graph notations. 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. We then show in Section 15.7.2 how heuristic optimization rules are applied to convert a 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 one. We also discuss the equivalence of various relational algebra expressions. Finally, Section 15.7.3 discusses the generation of query execution plans.

15.7.1 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 execution terminates when the root node is executed and produces the result relation for the query.

Figure 15.4a shows a query tree for query Q2 of Chapters 5 to 8: For every project located in 'Stafford', retrieve the project number, the controlling department number,

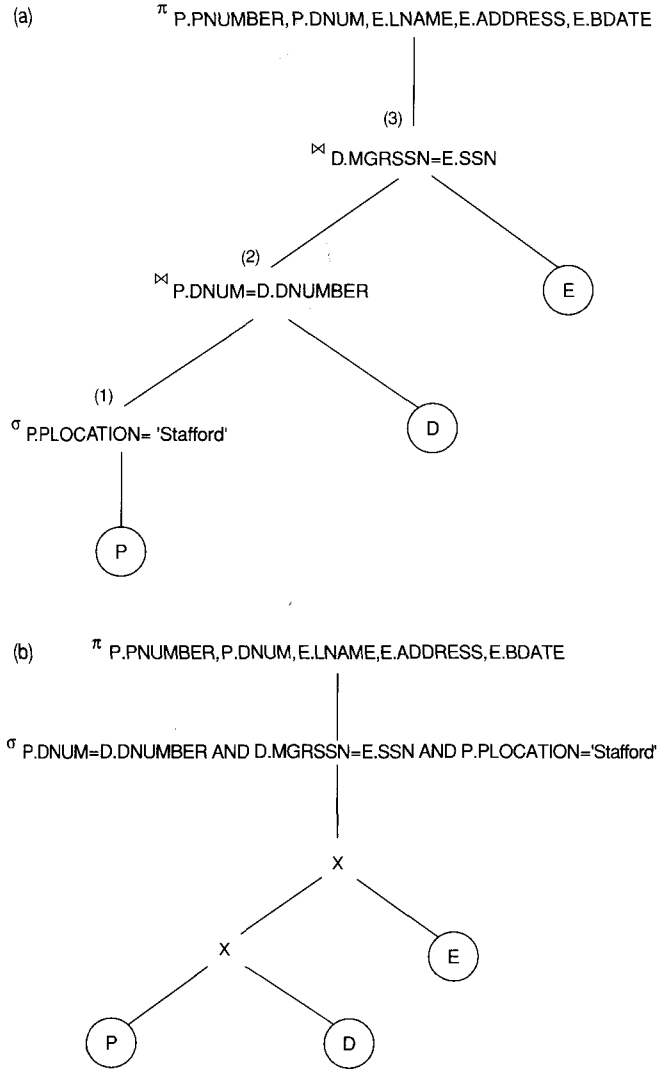


FIGURE 15.4 Two query trees for the query Q2. (a) Query tree corresponding to the relational algebra expression for Q2. (b) Initial (canonical) query tree for SQL query Q2.

and the department manager's last name, address, and birthdate. This query is specified on the relational schema of Figure 5.5 and corresponds to the following relational algebra expression:

$$\pi_{\text{PNUMBER, DNUM, LNAME, ADDRESS, BDATE}} ((\sigma_{\text{PLOCATION='STAFFORD'}}(\text{PROJECT})) \bowtie_{\text{DNUM=DNUMBER}} (\text{DEPARTMENT})) \bowtie_{\text{MGRSSN=SSN}} (\text{EMPLOYEE}))$$

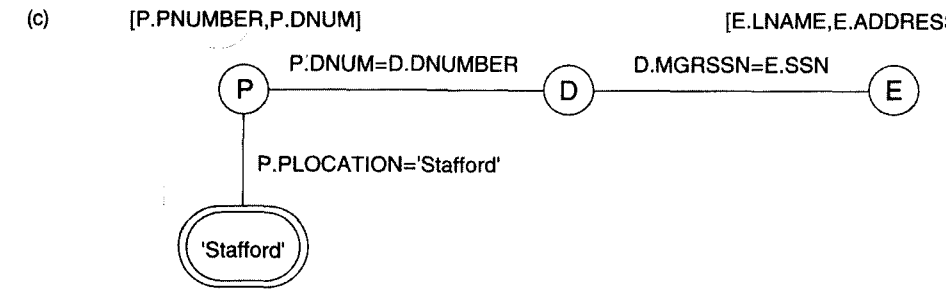


FIGURE 15.4(CONTINUED) (c) Query graph for Q2.

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.MGRSSN=E.SSN AND
           P.PLOCATION='STAFFORD';
```

In Figure 15.4a the three relations PROJECT, DEPARTMENT, and EMPLOYEE are represented by leaf nodes P, D, and E, while the relational algebra operations of the expression are represented by internal tree nodes. When this query tree is executed, the node marked (1) in Figure 15.4a 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 representation of a query is the **query graph** notation. Figure 15.4c 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 15.4c. 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.

15. Hence, a query graph corresponds to a *relational calculus* expression (see Chapter 6).

15.7.2 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 correspond to 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 15.4b. 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 (×) operations. For example, if the PROJECT, DEPARTMENT, and EMPLOYEE relations had record sizes of 100, 50, and 150 bytes and contained 100, 20, and 5000 tuples, respectively, the result of the CARTESIAN PRODUCT would contain 10 million tuples of record size 300 bytes each. However, the query tree in Figure 15.4b is in a simple standard form that can be easily created. It is now the job of the heuristic query optimizer to transform this initial query tree into a **final query tree** that is efficient to execute.

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

Example of Transforming a Query. Consider the following query Q on the database of Figure 5.5: "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:

```
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 15.5a. Executing this tree directly first creates a very large file containing the CARTESIAN PRODUCT of the entire EMPLOYEE, WORKS_ON, and PROJECT files. However, this 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 15.5b 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 15.5c. This uses the information that PNUMBER is a key attribute of the project relation, and hence the SELECT operation on the

16. A query may also be stated in various ways in a high-level query language such as SQL (see Chapter 8).

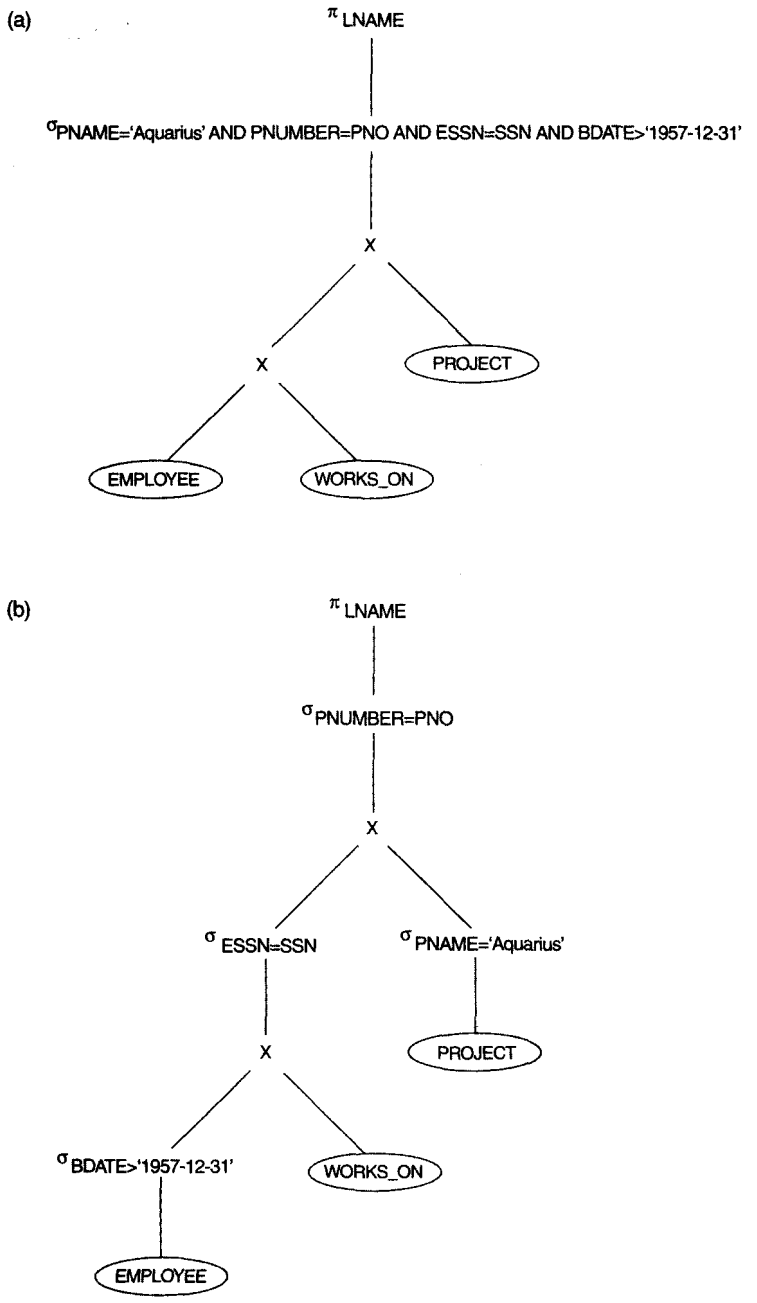


FIGURE 15.5 Steps in converting a query tree during heuristic optimization. (a) Initial (canonical) query tree for SQL query Q. (b) Moving SELECT operations down the query tree.

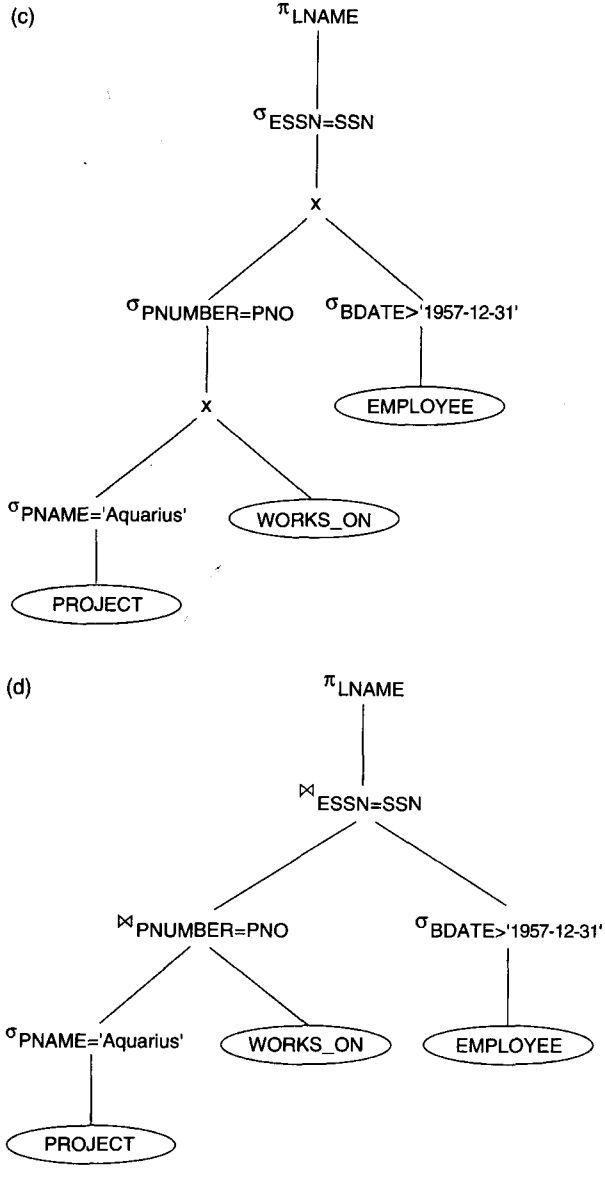


FIGURE 15.5(CONTINUED) Steps in converting a query tree during heuristic optimization. (c) Applying the more restrictive SELECT operation first. (d) Replacing CARTESIAN PRODUCT and SELECT with JOIN operations.

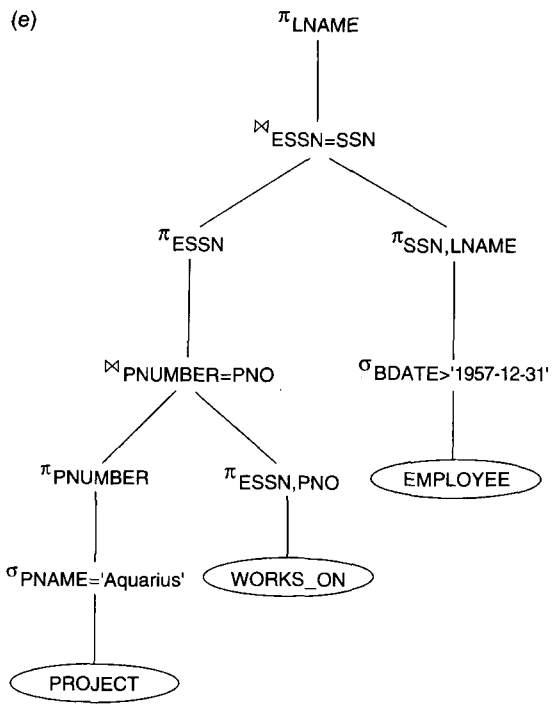


FIGURE 15.5(CONTINUED) Steps in converting a query tree during heuristic optimization. (e) Moving PROJECT operations down the query tree.

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 15.5d. 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 15.5e. 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 another 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. Here 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 equivalent. In Section 5.1.2 we gave an alternative definition of *relation* that makes order of attributes unimportant; we will use this definition here. We now 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:

   $\sigma_{c1 \text{ AND } c2 \text{ AND } \ldots \text{ AND } cn}(R) \equiv \sigma_{c1}(\sigma_{c2}(\ldots(\sigma_{cn}(R))\ldots))$

2. Commutativity of σ: The σ operation is commutative:

   $\sigma_{c1}(\sigma_{c2}(R)) \equiv \sigma_{c2}(\sigma_{c1}(R))$

3. Cascade of π: In a cascade (sequence) of π operations, all but the last one can be ignored:

   $\pi_{List1}(\pi_{List2}(\ldots(\pi_{Listn}(R))\ldots)) \equiv \pi_{List1}(R)$

4. Commuting σ with π: If the selection condition c involves only those attributes A1, ..., An in the projection list, the two operations can be commuted:

   $\pi_{A1, A2, \ldots, An}(\sigma_c(R)) \equiv \sigma_c(\pi_{A1, A2, \ldots, An}(R))$

5. Commutativity of $\bowtie$ (and $\times$): The $\bowtie$ 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 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 (c1 AND c2), where condition c1 involves only the attributes of R and condition $c2$ involves only the attributes of S, the operations commute as follows:

   $\sigma_c(R \bowtie S) \equiv (\sigma_{c1}(R)) \bowtie (\sigma_{c2}(S))$

   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 $L = \{A_1, \ldots, A_n, B_1, \ldots, B_m\}$, where $A_1, \ldots, A_n$ are attributes of R and $B_1, \ldots, B_m$ are attributes of S. If the join condition c involves only attributes in L, the two operations can be commuted as follows:

   $\pi_L(R \bowtie_c S) \equiv (\pi_{A1, \ldots, An}(R)) \bowtie_c (\pi_{B1, \ldots, Bm}(S))$

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 $A_{n+1}, \ldots, A_{n+k}$ of R and $B_{m+1}, \ldots, B_{m+p}$ of S are involved in the join condition c but are not in the projection list L, the operations commute as follows:

$$\pi_L (R \bowtie_c S) \equiv \pi_L ((\pi_{A1, \ldots, An, An+1, \ldots, An+k}(R)) \bowtie_c (\pi_{B1, \ldots, Bm, Bm+1, \ldots, Bm+p} (S)))$$

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 rules (DeMorgan's laws):

```
NOT (c1 AND c2) ≡ (NOT c1) OR (NOT c2)
NOT (c1 OR c2) ≡ (NOT c1) AND (NOT c2)
```

Additional transformations discussed in Chapters 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 an optimized tree that is more efficient to execute (in most cases). The algorithm will lead to transformations similar to those discussed in our example of Figure 15.5. 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.
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 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 15.5(b) shows the tree of Figure 15.5(a) after applying steps 1 and 2 of the algorithm; Figure 15.5(c) shows the tree after step 3; Figure 15.5(d) after step 4; and Figure 15.5(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. We now summarize the basic 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. This is done by moving SELECT and PROJECT operations

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

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.

as far down the tree as possible. In addition, 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. This is done by reordering the leaf nodes of the tree among themselves while avoiding Cartesian products, and adjusting the rest of the tree appropriately.

15.7.3 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 5, whose corresponding relational algebra expression is

$$\pi_{\text{FNAME, LNAME, ADDRESS}}(\sigma_{\text{DNAME='RESEARCH'}}(\text{DEPARTMENT}) \bowtie_{\text{DNUMBER=DNO}} \text{EMPLOYEE})$$

The query tree is shown in Figure 15.6. To convert this into an execution plan, the optimizer might choose an index search for the SELECT operation (assuming one exists), a table scan as access method for EMPLOYEE, a nested-loop join algorithm for the join, and a scan of the JOIN result for the PROJECT operator. In addition, the approach taken for executing the query may specify a materialized or a pipelined evaluation.

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

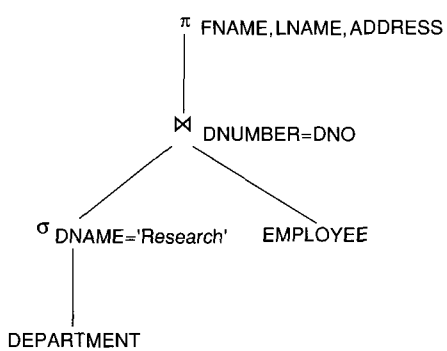


FIGURE 15.6 A query tree for query Q1.

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.

15.8 Using Selectivity and Cost Estimates in Query Optimization

A query optimizer should not depend solely on heuristic rules; it should also estimate and compare the costs of executing a query using different execution strategies and should choose the strategy with the *lowest cost estimate*. For this approach to work, accurate cost estimates are required so that different strategies are compared fairly and realistically. In addition, we 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 15.1 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.

We call this approach **cost-based query optimization,**[19] and 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 one. In Section 15.8.1 we discuss the components of query execution cost. In Section 15.8.2 we discuss the type of information needed in cost functions. This information is kept in the DBMS catalog. In Section 15.8.3 we give examples of cost functions for the SELECT operation, and in Section 15.8.4 we discuss cost functions for two-way JOIN operations. Section 15.8.5 discusses multiway joins, and Section 15.8.6 gives an example.

15.8.1 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 searching for, reading, and writing data blocks that reside on secondary storage, mainly on disk. The cost of searching for records in a file depends on the type of access structures on that file, such as ordering, hashing, and primary or secondary indexes. In addition, factors

19. This approach was first used in the optimizer for the SYSTEM R experimental DBMS developed at IBM.

such as whether the file blocks are allocated contiguously on the same disk cylinder or scattered on the disk affect the access cost.

2. *Storage cost:* This is the cost of storing 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 data buffers during query execution. Such operations include searching for and sorting records, merging records for a join, and performing computations on field values.
4. *Memory usage cost:* This is the cost pertaining to the number of 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.

For large databases, the main emphasis is 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 and main memory. 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.

15.8.2 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 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 access method* and the *primary access attributes* for each file. The file records may be unordered, ordered by an attribute with or without a primary or clustering index, or hashed on a key attribute. Information is kept on all secondary indexes and 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 its **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. In the next two sections we examine how some of these parameters are used in cost functions for a cost-based query optimizer.

15.8.3 Examples of Cost Functions for SELECT

We now give cost functions for the selection algorithms S1 to S8 discussed in Section 15.3.1 in terms of *number of block transfers* between memory and disk. These cost functions are estimates that ignore computation time, storage cost, and other factors. The cost for method Si 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, only half the file blocks are searched on the average before finding the record, so $C_{S1b} = (b/2)$ if the record is found; if no record 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.
- S3. *Using a primary index (S3a) or hash key (S3b) to retrieve a single record:* For a primary index, retrieve one more block than the number of index levels; hence, $C_{S3a} = x + 1$. For hashing, the cost function is approximately $C_{S3b} = 1$ for static hashing or linear hashing, and it is 2 for extendible hashing (see Chapter 13).
- 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.
- S5. *Using a clustering index to retrieve multiple records:* Given an equality condition, 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 accessed, giving $C_{S5} = x + \lceil (s/bfr) \rceil$.
- S6. *Using a secondary (B^+-tree) index:* On an *equality* comparison, s records will satisfy the condition, where s is the selection cardinality of the indexing attribute.

20. As we mentioned earlier, more accurate optimizers may store histograms of the distribution of records over the data values for an attribute.

However, because the index is nonclustering, each of the records may reside on a different block, so the (worst case) cost estimate is $C_{S6a} = x + s$. This reduces to $x + 1$ for a key indexing attribute. 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.

- 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 memory buffer whether each retrieved record satisfies the remaining conditions in the conjunction.
- S8. *Conjunctive selection using a composite index:* Same as S3a, S5, or S6a, 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 of Figure 5.5 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 $x_{SALARY} = 3$ and average selection cardinality $s_{SALARY} = 20$.
2. A secondary index on the key attribute SSN, with $x_{SSN} = 4$ ($s_{SSN} = 1$).
3. A secondary index on the nonkey attribute DNO, with $x_{DNO} = 2$ and first-level index blocks $b_{I1DNO} = 4$. There are $d_{DNO} = 125$ distinct values for DNO, so the selection cardinality of DNO is $s_{DNO} = (r_E/d_{DNO}) = 80$.
4. A secondary index on SEX, with $x_{SEX} = 1$. There are $d_{SEX} = 2$ values for the sex attribute, so the average selection cardinality is $s_{SEX} = (r_E/d_{SEX}) = 5000$.

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

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

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

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

(OP4): $\sigma_{DNO=5\ AND\ SALARY>30000\ AND\ SEX='F'}$(EMPLOYEE)

The cost of the brute force (linear search) 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 S1 or method S6a; the cost estimate for S6a is $C_{S6a} = x_{SSN} + 1 = 4 + 1 = 5$, and it is chosen over Method S1, whose average cost is $C_{S1b} = 1000$. For OP2 we can use either method S1 (with estimated cost $C_{S1a} = 2000$) or method S6*b* (with estimated cost $C_{S6b} = x_{DNO} + (b_{I1DNO}/2) + (r_E/2) = 2$

+ (4/2) + (10,000/2) = 5004), so we choose the brute force approach for OP2. For OP3 we can use either method S1 (with estimated cost C_{S1a} = 2000) or method S6*a* (with estimated cost $C_{S6a} = x_{DNO} + s_{DNO} = 2 + 80 = 82$), 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 brute force 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 $C_{S4} = x_{SALARY} + (b_E/2) = 3 + (2000/2) = 1003$. Using the condition (SEX = 'F') first gives a cost estimate $C_{S6a} = x_{SEX} + s_{SEX} = 1 + 5000 = 5001$. 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.

15.8.4 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, $0 \leq js \leq 1$. 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 $|(R \bowtie_c S)| \leq |S|$, so $js \leq (1/|R|)$.
2. If B is a key of S, then $|(R \bowtie_c S)| \leq |R|$, so $js \leq (1/|S|)$.

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 15.3.2. 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 discussed in Section 15.3.2.

- 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

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

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

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

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.

- 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 15.2 to estimate the sorting cost.

Example of Using the Cost Functions. Suppose that we have the EMPLOYEE file described in the example of the previous section, and assume that the DEPARTMENT file of Figure 5.5 consists of $r_D = 125$ records stored in $b_D = 13$ disk blocks. Consider the join operations

(OP6): EMPLOYEE $\bowtie_{DNO=DNUMBER}$ DEPARTMENT
(OP7): DEPARTMENT $\bowtie_{MGRSSN=SSN}$ EMPLOYEE

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.

Suppose that we have a primary index on DNUMBER of DEPARTMENT with $x_{DNUMBER} = 1$ level and a secondary index on MGRSSN of DEPARTMENT with selection cardinality $s_{MGRSSN} = 1$ and levels $x_{MGRSSN} = 2$. Assume that the join selectivity for OP6 is $js_{OP6} = (1/|\text{DEPARTMENT}|) = 1/125$ because DNUMBER is a key of DEPARTMENT. Also assume that the blocking factor for the resulting join file $bfr_{ED} = 4$ 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:

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

2. Using Method J1 with DEPARTMENT as outer loop:

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

3. Using Method J2 with EMPLOYEE as outer loop:

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

4. Using Method J2 with DEPARTMENT as outer loop:

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

Case 4 has the lowest cost estimate and will be chosen. Notice that if 15 memory buffers (or more) were available for executing the join instead of just three, 13 of them could be used to hold the entire DEPARTMENT relation in memory, one could be used as buffer for the result, and the cost for Case 2 could be drastically reduced to just $b_E + b_D + ((js_{OP6} * r_E * r_D)/bfr_{ED})$ or 4513, as discussed in Section 15.3.2. As an exercise, the reader should perform a similar analysis for OP7.

15.8.5 Multiple Relation Queries and Join Ordering

The algebraic transformation rules in Section 15.7.2 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. In general, a query that joins n relations will 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 where 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 15.7. (Note that the trees in Figure 15.5 are also left-deep trees.)

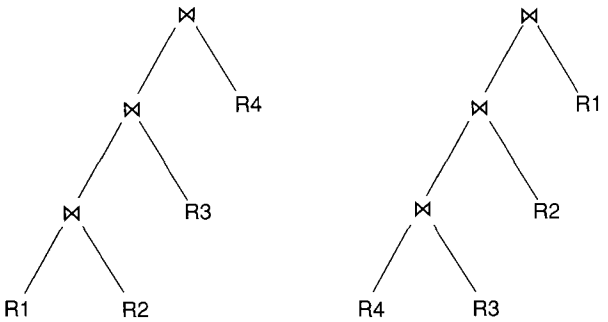


FIGURE 15.7 *Two left-deep (join) query trees.*

With left-deep trees, the right child is considered to be the inner relation when executing a nested-loop join. One advantage of left-deep (or right-deep) trees is that they are amenable to pipelining, as discussed in Section 15.6. For instance, consider the first left-deep tree in Figure 15.7 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 matching tuples. As a resulting block of tuples is produced from the join of R1 and R2, it could be used to probe R3. Likewise, as a resulting page of tuples is produced from this join, it 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 Section 15.6), 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.

15.8.6 Example to Illustrate Cost-Based Query Optimization

We will consider query Q2 and its query tree shown in Figure 15.4a to illustrate cost-based query optimization:

```
Q2: SELECT PNUMBER, DNUM, LNAME, ADDRESS, BDATE
    FROM   PROJECT, DEPARTMENT, EMPLOYEE
    WHERE  DNUM=DNUMBER AND MGRSSN=SSN AND PLOCATION='STAFFORD';
```

Suppose we have the statistical information about the relations shown in Figure 15.8. The LOW_VALUE and HIGH_VALUE statistics have been normalized for clarity. The tree in Figure 15.4a 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).

(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	MGRSSN	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	UNIQUENES	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.

FIGURE 15.8 Sample statistical information for relations in Q2. (a) Column information. (b) Table information. (c) Index information.

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 15.8, 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. The statistical information on the PROJ_PLOC index (see Figure 15.8) 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 15.8 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 15.8) 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 MGRSSN 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.

15.9 Overview of Query Optimization in ORACLE

The ORACLE DBMS (Version 7) 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 (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 (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 lowest estimated cost. The estimated query cost is proportional to the expected elapsed time needed to execute the query with the given execution plan. 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.[22] 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 5.5. 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 is a good example of the sophisticated approach taken to optimize SQL queries in commercial RDBMSs.

15.10 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 only illustrate it with a simple example. Consider the SQL query:

```
SELECT E.LNAME, M.LNAME
FROM   EMPLOYEE AS E, EMPLOYEE AS M
WHERE  E.SUPERSSN=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

22. Such hints have also been called query *annotations*.

can earn more than his or her direct supervisor. If the semantic query optimizer checks for the existence of this constraint, it need not 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 query and that may semantically optimize it can also be quite time-consuming. With the inclusion of active rules in database systems (see Chapter 24), semantic query optimization techniques may eventually be fully incorporated into the DBMSs of the future.

15.11 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 then 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

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

15.2. 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.

15.3. What is a query execution plan?

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

15.5. 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.
15.6. How many different join orders are there for a query that joins 10 relations?
15.7. What is meant by *cost-based query optimization*?
15.8. What is the difference between *pipelining* and *materialization*?
15.9. 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?
15.10. Discuss the different types of parameters that are used in cost functions. Where is this information kept?
15.11. List the cost functions for the SELECT and JOIN methods discussed in Section 15.8.
15.12. What is meant by semantic query optimization? How does it differ from other query optimization techniques?

Exercises

15.13. Consider SQL queries Q1, Q8, Q1B, Q4, and Q27 from Chapter 8.
   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, then show how the query tree is optimized by the algorithm outlined in Section 15.7.
   c. For each query, compare your own query trees of part (a) and the initial and final query trees of part (b).
15.14. 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?
15.15. Develop cost functions for the PROJECT, UNION, INTERSECTION, SET DIFFERENCE, and CARTESIAN PRODUCT algorithms discussed in Section 15.4.
15.16. 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.
15.17. Can a nondense index be used in the implementation of an aggregate operator? Why or why not?
15.18. Calculate the cost functions for different options of executing the JOIN operation OP7 discussed in Section 15.3.2.
15.19. 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.
15.20. Estimate the cost of operations OP6 and OP7, using the formulas developed in Exercise 15.9.
15.21. Extend the sort-merge join algorithm to implement the left outer join operation.
15.22. Compare the cost of two different query plans for the following query:

$$\sigma_{SALARY > 40000}(EMPLOYEE \bowtie_{DNO=DNUMBER} DEPARTMENT)$$

Use the database statistics in Figure 15.8.

Selected Bibliography

A survey by Graefe (1993) discusses query execution in database systems and includes an extensive bibliography. A survey paper by Jarke and Koch (1984) gives a taxonomy of query optimization and includes a bibliography of work in this area. 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.

Whang (1985) discusses query optimization in OBE (Office-By-Example), which is a system based on QBE. Cost-based optimization was introduced in the SYSTEM R experimental DBMS and is discussed in Astrahan et al. (1976). Selinger et al. (1979) discuss the 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). More recent work on semantic query optimization is reported in Chakravarthy et al. (1990), Shenoy and Ozsoyoglu (1989), and Siegel et al. (1992).

16

Practical Database Design and Tuning

In this chapter, we first discuss the issues that arise in physical database design in Section 16.1. Then, we discuss how to improve database performance through database tuning in Section 16.2.

16.1 Physical Database Design in Relational Databases

In this section we first discuss the physical design factors that affect the performance of applications and transactions; we then comment on the specific guidelines for RDBMSs.

16.1.1 Factors That Influence Physical Database Design

Physical design is an activity where the goal is not only to come up with the appropriate structuring of data in storage but 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 we know the queries, transactions, and applications that are expected to run on the database. We must analyze these applications, their expected frequencies of invocation,

any time constraints on their execution, and the expected frequency of update operations. We discuss each of these factors next.

A. Analyzing the Database Queries and Transactions. Before undertaking physical database design, we must have a good idea of the intended use of the database by defining the queries and transactions that we expect to run on the database in a high-level form. For each query, we should specify the following:

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. The attributes on which any join conditions or conditions to link multiple tables or objects for the query are specified.
4. The attributes whose values will be retrieved by the query.

The attributes listed in items 2 and 3 above are candidates for definition of access structures. For each update transaction or operation, we should specify the following:

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 previously in item 3 are candidates for access structures. 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 queries and 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 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, over all the queries and transactions. Generally, for large volumes of processing, the informal "80–20 rule" applies, which states that 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

1. For simplicity we use the term *files*. This can be substituted by tables or classes or objects.

occasions when it is invoked and that it should never take more than 20 seconds. Such performance 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.

D. Analyzing the Expected Frequencies of Update Operations. A minimum number of access paths should be specified for a file that is updated frequently, because updating the access paths themselves slows down the update 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 or constrained to be unique. The existence of an index (or other access path) makes it sufficient to search only the index when checking this constraint, since all values of the attribute will exist in the leaf nodes of the index.

Once we have compiled the preceding information, we can address the physical database design decisions, which consist mainly of deciding on the storage structures and access paths for the database files.

16.1.2 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 file for each relation and the attributes on which indexes should be defined. At most one of the indexes on each file may be a primary or clustering index. Any number of additional secondary indexes can be created.[2]

Design Decisions about Indexing. The attributes whose values are required in equality or range conditions (selection operation) and those that are keys or that participate in join conditions (join operation) require access paths.

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, 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 attribute must be a key, or there must be some query that uses that attribute either in a selection condition (equality or range of values) or in a join. One factor in favor of setting up many indexes is that some queries can be processed by just scanning the indexes without retrieving any data.

2. The reader should review the various types of indexes described in Section 13.1. For a clearer understanding of this discussion, it is also useful to be familiar with the algorithms for query processing discussed in Chapter 15.

2. *What attribute or attributes to index on:* An index can be constructed on one or multiple attributes. If multiple attributes from one relation are involved together in several queries, (for example, `(garment_style_#, color)` in a garment inventory database), a multiattribute index is warranted. The ordering of attributes within a multiattribute index must correspond to the queries. For example, 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.) If a table requires several indexes, the decision about which one should be a clustered 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.
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 14). 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).
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 13.9 would be suitable. Currently, they are not offered by most commercial RDBMSs.

Denormalization as a Design Decision for Speeding Up Queries. The ultimate goal during normalization (see Chapters 10 and 11) was to separate the logically related 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 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 adds to a table attributes that are needed for answering queries or producing reports so that a join with another table, which contains the newly added attribute, is avoided. This reintroduces a partial functional dependency or a transitive dependency into the table, thereby creating the associated redundancy problems (see Chapter 10).

Other forms of denormalization consist of storing extra tables to maintain original functional dependencies that are lost during a BCNF decomposition. For example, Figure 10.13 showed 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 consider T1 and T2 as updatable base tables whereas TEACH can be created as a view.

16.2 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 16.1.1 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. It is therefore necessary to monitor and revise the physical database design constantly. The goals of tuning are as follows:

- To make applications run faster.
- To lower the response time of queries/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 16.1.3 are revisited during the tuning phase, which is a continued adjustment of design. We give only a brief overview of the tuning process below.[3] The inputs to the tuning process include statistics related to the factors mentioned in Section 16.1.1. 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/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).

3. Interested readers should consult Shasha (1992) for a detailed discussion of tuning.

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 ports.
- *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, optimization times during query optimization.
- *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, etc.

Many of the above statistics relate to transactions, concurrency control, and recovery, which are to be discussed in Chapters 17 through 19. 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 overheard of logging and unnecessary dumping of data.
- How to optimize 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 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 problems of tuning for the specific DBMS. We briefly discuss the tuning of various physical database design decisions below.

16.2.1 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 be causing excessive overhead 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 were used. By analyzing these execution plans,

4. The reader will need to look ahead and review Chapters 17–19 for explanation of these terms.

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 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 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. Just for illustration, consider the sparse and dense indexes of Chapter 14. Sparse indexes have one index pointer for each page (disk block) in the data file; dense indexes 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.

16.2.2 Tuning the Database Design

We already discussed in Section 16.1.2 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, we may revert to the logical database design, make adjustments to the logical schema, and remap it to a new set of physical tables and indexes.

As we pointed out 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. One may be replaced by the other.

5. Note that 3NF and 2NF address different types of problem dependencies which are independent of each other; hence the normalization (or denormalization) order between them is arbitrary.

- 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 into 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. Each table groups 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 had a very large number of rows (say 100,000) and queries about phone numbers and salary information are totally distinct, this separation of tables may work better. This is also called **vertical partitioning.**
- Attribute(s) from one table may be repeated in another even though this creates redundancy and a potential anomaly. For example, Partname may be replicated in tables wherever the Part# appears (as foreign key), but there may be one master table called PART_MASTER(Part#, Partname, ...) 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 queries or transactions, with or without sacrificing the normal forms, are commonplace in practice.

16.2.3 Tuning Queries

We already discussed how query performance is dependent upon 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 query:

```
SELECT SSN FROM EMPLOYEE
WHERE DNO IN (SELECT DNUMBER FROM DEPARTMENT
              WHERE MGRSSN = '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 far 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 use of correlated queries, temporaries are useful. Consider the query:

```
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 it more efficient, it can be broken into two queries where the first query just computes the maximum salary in each department as follows:

```
SELECT MAX (SALARY) AS HIGHSALARY, DNO INTO TEMP
FROM EMPLOYEE
GROUP BY DNO;

SELECT SSN
FROM EMPLOYEE, TEMP
WHERE SALARY = HIGHSALARY AND EMPLOYEE.DNO = TEMP.DNO;
```

6. If multiple options for 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 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 inner query.
   - Uncorrelated subqueries without aggregates.
   - Correlated subqueries with aggregates in inner query.
   - Correlated subqueries without aggregates.

   Out of the above four types, 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 (2) above, most query optimizers may not use an index on DNO in EMPLOYEE. 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 an overkill, because a query may be posed directly against a base table, rather than going through a view that is defined by a join.

16.2.4 Additional Query Tuning Guidelines

Additional techniques for improving queries apply in certain situations:

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 in expediting 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 (1992).

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 ((PRODUCT_TYPE BETWEEN 1 AND 3) OR (PRODUCT_
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 PRODUCT_TYPE. Instead, using

```
SELECT REGION#, PROD_TYPE, MONTH, SALES
FROM SALES_STATISTICS
WHERE (REGION# = 3 AND (PRODUCT_TYPE BETWEEN 1 AND 3)) OR (REGION# =
3 AND (PRODUCT_TYPE BETWEEN 8 AND 10));
```

can use a composite index on (REGION#, PRODUCT_TYPE) and work much more efficiently.

We have covered in this section most of the common opportunities where inefficiency of a query may be corrected by some simple corrective action such as using a temporary, avoiding certain types of constructs, or avoiding use of views. The problems and the remedies will depend upon the workings of a query optimizer within an RDBMS. Detailed literature exists in terms of individual manuals on database tuning guidelines for database administration by the RDBMS vendors.

16.3 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, modifications of indexing, and changes to queries as a part of database tuning.

Review Questions

16.1. What are the important factors that influence physical database design?
16.2. Discuss the decisions made during physical database design.
16.3. Discuss the guidelines for physical database design in RDBMSs.
16.4. Discuss the types of modifications that may be applied to the logical database design of a relational database.
16.5. Under what situations would denormalization of a database schema be used? Give examples of denormalization.
16.6. Discuss the tuning of indexes for relational databases.
16.7. Discuss the considerations for reevaluating and modifying SQL queries.
16.8. Illustrate the types of changes to SQL queries that may be worth considering for improving the performance during database tuning.
16.9. What functions do the typical database design tools provide?

Selected Bibliography

Wiederhold (1986) covers all phases of database design, with an emphasis on physical design. O'Neil (1994) 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.