chapter 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. After discussing different storage technologies, we 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 file records on disk. Section 13.5 discusses the various types of operations that are typically applied to file records. We present three primary methods for organizing file records on disk: unordered records, in Section 13.6; ordered records, in Section 13.7; and hashed records, in Section 13.8.

Section 13.9 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 13.10 describes RAID (Redundant Arrays of Inexpensive (or Independent) Disks)—a data storage system architecture that is commonly used in large organizations for better reliability and performance. Finally, in Section 13.11 we describe two recent developments in the storage systems area: storage area networks (SAN) and network attached storage (NAS). In Chapter 14 we discuss techniques for creating auxiliary data structures, called indexes, which speed up the search for and retrieval of records. These techniques involve storage of auxiliary data, called index files, in addition to the file records themselves.

Chapters 13 and 14 may be browsed through or even omitted by readers who have already studied file organizations. The material covered here, in particular Sections 13.1 through 13.8, is necessary for understanding Chapters 15 and 16, which 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 and tertiary storage.** This category includes magnetic disks, optical disks, and tapes. Hard disk drives are classified as secondary storage, whereas removable media are considered tertiary storage. These devices usually have a larger capacity, cost less, and provide slower access to data than do primary storage devices. Data in secondary or tertiary storage cannot be processed directly by the CPU; first it must be copied into primary storage.

We will give an overview of the various storage devices used for primary and secondary storage in Section 13.1.1 and then we will 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. It is popularly called **main memory**. The advantage of DRAM is its low cost, which continues to decrease; the drawback is its volatility[1] and lower speed compared with static RAM. At the *secondary and ter-*

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

tiary storage level, the hierarchy includes magnetic disks, as well as **mass storage** in the form of CD-ROM (Compact Disk–Read-Only Memory) and DVD devices, and finally tapes at the least expensive end of the hierarchy. The **storage capacity** is measured in kilobytes (Kbyte or 1000 bytes), megabytes (MB or 1 million bytes), gigabytes (GB or 1 billion bytes), and even terabytes (1000 GB).

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 part of the database into main memory. Eight to 16 GB of RAM on a single server is becoming commonplace. In some cases, entire databases can be kept in main memory (with a backup copy on magnetic disk), leading to **main memory databases**; these are particularly useful in real-time applications that require extremely fast response times. An example is telephone switching applications, which store databases that contain routing and line information in main memory.

Between DRAM and magnetic disk storage, another form of memory, **flash memory**, is becoming common, particularly because it is nonvolatile. Flash memories are high-density, high-performance memories using EEPROM (Electrically Erasable Programmable Read-Only Memory) technology. The advantage of flash memory is the fast access speed; the disadvantage is that an entire block must be erased and written over simultaneously.[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, and so on.

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

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 automat-

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).

ically loaded onto tape drives—are becoming popular as **tertiary storage** to hold terabytes of data. For example, NASA's EOS (Earth Observation Satellite) system stores archived databases in this fashion.

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

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 secondary storage than for primary storage.

Some of the newer technologies—such as optical disks, DVDs, and tape jukeboxes—are likely to provide viable alternatives to the use of magnetic disks. In the future, databases may therefore reside at different levels of the memory hierarchy from those described in Section 13.1.1. However, it is anticipated that magnetic disks will continue to be the primary medium of choice for large databases for years to come. Hence, it is important to study and understand the properties and characteristics of magnetic disks and the way data files can be organized on disk in order to design effective databases with acceptable performance.

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

The techniques used to store large amounts of structured data on disk are important for database designers, the DBA, and implementers of a DBMS. Database designers and the DBA must know the advantages and disadvantages of each storage technique when they design, implement, and operate a database on a specific DBMS. Usually, the DBMS has several options available for organizing the data. The process of **physical database design** involves choosing the particular data organization techniques that best suit the given application requirements from among the options. DBMS system implementers must study data organization techniques so that they can implement them efficiently and thus provide the DBA and users of the DBMS with sufficient options.

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

There are several **primary file organizations**, which determine how the file records are *physically placed* on the disk, *and hence how the records can be accessed.* A *heap file* (or *unordered file*) places the records on disk in no particular order by appending new records at the end of the file, whereas a *sorted file* (or *sequential file*) keeps the records ordered by the value of a particular field (called the sort key). A *hashed file* uses a hash function applied to a particular field (called the hash key) to determine a record's placement on disk. Other primary file organizations, such as *B-trees,* use tree structures. We discuss primary file organizations in Sections 13.6 through 13.9. A **secondary organization** or **auxiliary access structure** allows efficient access to file records based on *alternate fields* than those that have been used for the primary file organization. Most of these exist as indexes and will be discussed in Chapter 14.

13.2 Secondary Storage Devices

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

13.2.1 Hardware Description of Disk Devices

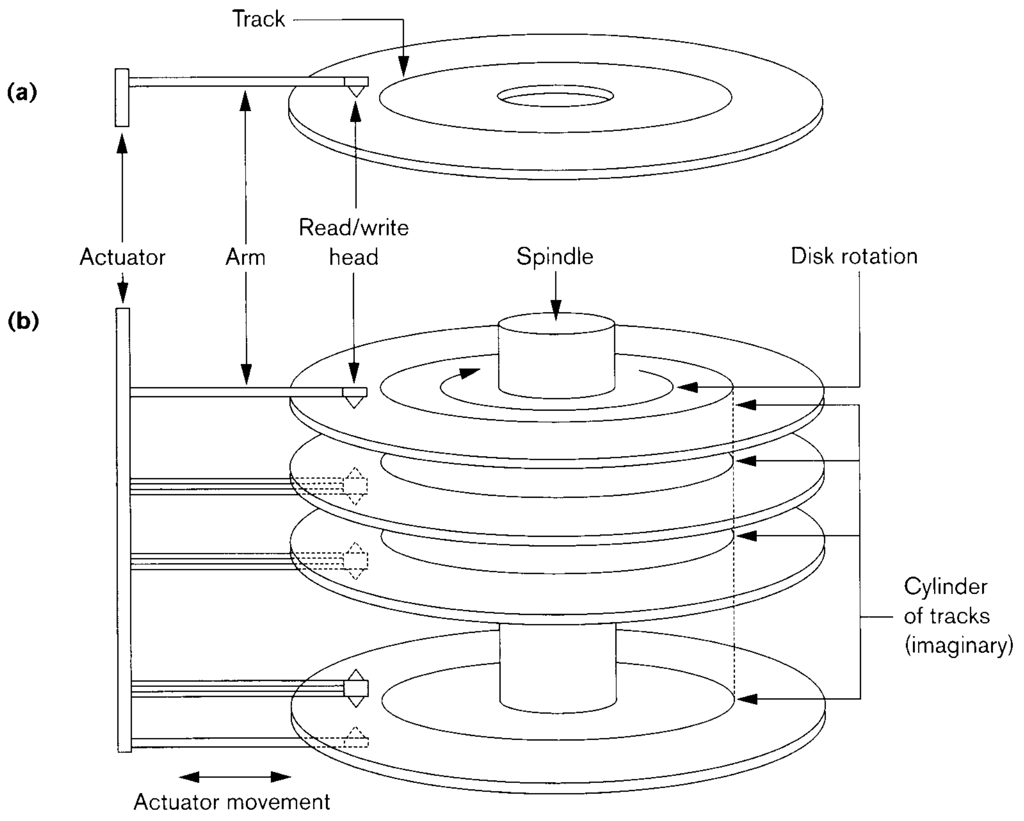
Magnetic disks are used for storing large amounts of data. The most basic unit of data on the disk is a single **bit** of information. By magnetizing an area on disk in certain ways, one can make it represent a bit value of either 0 (zero) or 1 (one). To code information, bits are grouped into **bytes** (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 KB to 1.5 MB; hard disks for micros typically hold from several hundred MB up to tens of GB; and large disk packs used with servers and mainframes have capacities of hundreds of GB. Disk capacities continue to grow as technology improves.

Whatever their capacity, all disks are made of magnetic material shaped as a thin circular disk, as shown in Figure 13.1(a), and protected by a plastic or acrylic cover. A disk is **single-sided** if it stores information on one of its surfaces only and **double-**

sided if both surfaces are used. To increase storage capacity, disks are assembled into a **disk pack**, as shown in Figure 13.1(b), which may include many disks and therefore many surfaces. Information is stored on a disk surface in concentric circles of *small width*,[4] each having a distinct diameter. Each circle is called a **track**. In disk packs, tracks with the same diameter on the various surfaces are called a **cylinder** because of the shape they would form if connected in space. The concept of a cylinder is important because data stored on one cylinder can be retrieved much faster than if it were distributed among different cylinders.

The number of tracks on a disk ranges from a few hundred to a few thousand, and the capacity of each track typically ranges from tens of Kbytes to 150 Kbytes. Because a track usually contains a large amount of information, it is divided into smaller blocks or sectors. The division of a track into **sectors** is hard-coded on the disk surface and cannot be changed. One type of sector organization, as shown in

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



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

Figure 13.2(a), calls a portion of a track that subtends a fixed angle at the center a sector. Several other sector organizations are possible, one of which is to have the sectors subtend smaller angles at the center as one moves away, thus maintaining a uniform density of recording, as shown in Figure 13.2(b). A technique called ZBR (Zone Bit Recording) allows a range of cylinders to have the same number of sectors per arc. For example, cylinders 0–99 may have one sector per track, 100–199 may have two per track, and so on. Not all disks have their tracks divided into sectors.

The division of a track into equal-sized **disk blocks** (or **pages**) is set by the operating system during disk **formatting** (or **initialization**). Block size is fixed during initialization and cannot be changed dynamically. Typical disk block sizes range from 512 to 8192 bytes. A disk with hard-coded sectors often has the sectors subdivided into blocks during initialization. Blocks are separated by fixed-size **interblock gaps**, which include specially coded control information written during disk initialization. This information is used to determine which block on the track follows each 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

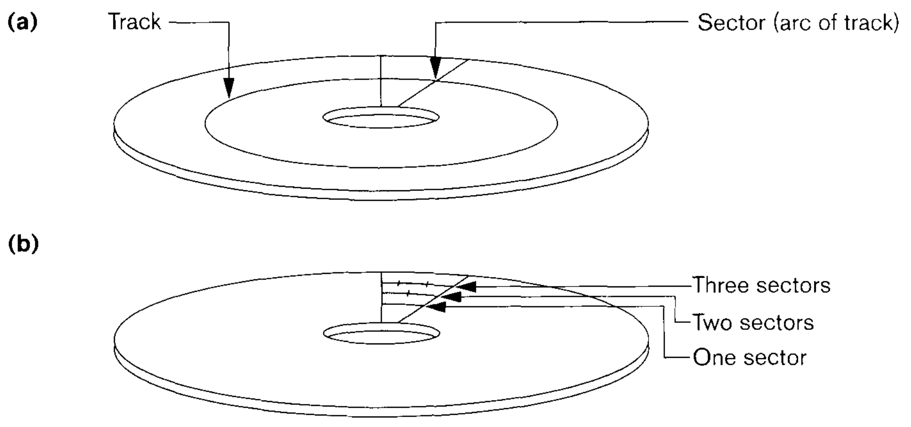


Figure 13.2
Different sector organizations on disk.
(a) Sectors subtending a fixed angle.
(b) Sectors maintaining a uniform recording density.

Table 13.1
Specifications of Typical High-end Cheetah Disks from Seagate

Description	**Cheetah 10K.6**	**Cheetah 10K.7**
Model Number	ST3146807LC	ST3300007LW
Form Factor (width)	3.5 inch	3.5 inch
Form Factor (height)	1 inch	1 inch
Height	25.4 mm	25.4 mm
Width	101.6 mm	101.6 mm
Length	146.05 mm	146.05 mm
Weight	0.73 Kg	0.726 kg
Capacity/Interface		
Formatted Capacity	146.8 Gbytes	300 Gbytes
Interface Type	80-pin	68-pin
Configuration		
Number of disks (physical)	4	4
Number of heads (physical)	8	8
Number of Cylinders	49,854	90,774
Total Tracks		726,192
Bytes per Sector	512	512
Areal Density	36,000 Mb/sq.inch	
Track Density	64,000 Tracks/inch	105,000 Tracks/inch
Recording Density	570,000 bits/inch	658,000 bits/inch
Bytes/Track (avg)		556
Performance		
Transfer Rates		
Internal Transfer Rate (min)	475 Mb/sec	472 Mb/sec
Internal Transfer Rate (max)	840 Mb/sec	944 Mb/sec
Formated Int. Transfer Rate (min)	43 MB/sec	59 MB/sec
Formated Int. Transfer Rate (max)	78 MB/sec	118 MB/sec
External I/O Transfer Rate (max)	320 MB/sec	320 MB/sec
Average Formatted Transfer Rate	59.9 MB/sec	59.5 MB/sec
Seek Times		
Avg. Seek Time (Read)	4.7 ms (typical)	4.7 ms (typical)
Avg. Seek Time (Write)	5.2 ms (typical)	5.3 ms (typical)
Track-to-track, Seek, Read	0.3 ms (typical)	0.2 ms (typical)
Track-to-track, Seek, Write	0.5 ms (typical)	0.5 ms (typical)
Full Disc Seek, Read		9.5 ms (typical)
Full Disc Seek, Write		10.3 ms (typical)
Average Latency	2.99 ms	3 msec
Other		
Default Buffer (cache) size	8,000 KB	8,192 KB
Spindle Speed	10000 RPM	10000 RPM
Power-on to Ready Time		25 sec

Table 13.1 (continued)
Specifications of Typical High-end Cheetah Disks from Seagate

Electrical Requirements	**Cheetah 10K.6**	**Cheetah 10K.7**
Current		
Typical Current (12VDC +/- 5%)	0.95 amps	1.09 amps
Typical Current (5VDC +/- 5%)	0.9 amps	0.68 amps
Idle Power (typ)	10.6 watts	10.14 watt
Reliability		
Mean Time Between Failure (MTBF)	1,200,000 Hours	1,400,000 Hours
Recoverable Read Errors	10 per 1012 bits	10 per 1012 bits read
Nonrecoverable Read Errors	10 per 1015 bits	10 per 1015 bits read
Seek Errors	10 per 108 bits	
Service Life	5 year(s)	5 year(s)
Limited Warranty Period	5 year(s)	5 year(s)

(Courtesy Seagate Technology)

block from disk is copied into the buffer; whereas for a **write** command, the contents of the buffer are copied into the disk block. Sometimes several contiguous blocks, called a **cluster**, may be transferred as a unit. In this case, the buffer size is adjusted to match the number of bytes in the cluster.

The actual hardware mechanism that reads or writes a block is the disk **read/write head**, which is part of a system called a **disk drive**. A disk or disk pack is mounted in the disk drive, which includes a motor that rotates the disks. A read/write head includes an electronic component attached to a **mechanical arm**. Disk packs with multiple surfaces are controlled by several read/write heads—one for each surface, as shown in Figure 13.1(b). All arms are connected to an **actuator** attached to another electrical motor, which moves the read/write heads in unison and positions them precisely over the cylinder of tracks specified in a block address.

Disk drives for hard disks rotate the disk pack continuously at a constant speed (typically ranging between 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 PCs 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 9 to 60 msec. For contiguous blocks, locating the first block takes from 9 to 60 msec, but transferring subsequent blocks may take only 0.4 to 2 msec each. Many search techniques take advantage of consecutive retrieval of blocks when searching for data on disk. In any case, a transfer time in the order of milliseconds is considered quite high compared with the time required to process data in main memory by current CPUs. Hence, locating data on disk is a *major bottleneck* in database applications. The file structures we discuss here and in Chapter 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, first we must scan the preceding $n - 1$ blocks. Data is stored on reels of high-capacity magnetic tape, somewhat similar to audiotapes or videotapes. A tape drive is required to read the data from or write the data to a **tape reel**. Usually, each group of bits that forms a byte is stored across the tape, and the bytes themselves are stored consecutively on the tape.

A read/write head is used to read or write data on tape. Data records on tape are also stored in blocks—although the blocks may be substantially larger than those for disks, and interblock gaps are also quite large. With typical tape densities of 1600 to 6250 bytes per inch, a typical interblock gap[5] of 0.6 inches corresponds to 960 to 3750 bytes of wasted storage space. It is customary to group many records together in one block for better space utilization.

The main characteristic of a tape is its requirement that we access the data blocks in **sequential order**. To get to a block in the middle of a reel of tape, the tape is mounted and then scanned until the required block gets under the read/write head. For this reason, tape access can be slow and tapes are not used to store online data, except for some specialized applications. However, tapes serve a very important function—**backing up** the database. One reason for backup is to keep copies of disk files in case the data is lost due to a disk crash, which can happen if the disk read/write head touches the disk surface because of mechanical malfunction. For this reason, disk files are copied periodically to tape. For many online critical applications, such as airline reservation systems, to avoid any downtime, mirrored systems are used to keep three sets of identical disks—two in online operation and one as backup. Here, offline disks become a backup device. The three are rotated so that they can be switched in case there is a failure on one of the live disk drives. Tapes can also be used to store excessively large database files. Database files that are seldom used or are outdated but required for historical record keeping can be **archived** on tape. Recently, smaller 8-mm magnetic tapes (similar to those used in camcorders) that can store up to 50 GB, as well as 4-mm helical scan data cartridges and writable CDs and DVDs, have become popular media for backing up data files from PCs and workstations. They are also used for storing images and system libraries. Backing up enterprise databases so that no transaction information is lost is a major undertaking. Currently, tape libraries with slots for several hundred cartridges are used with Digital and Superdigital Linear Tapes (DLTs and SDLTs) having capacities in hundreds of gigabytes that record data on linear tracks. Robotic arms are used to write on multiple cartridges in parallel using multiple tape drives with automatic labeling software to identify the backup cartridges. An example of a giant library is the L5500 model of Storage Technology that can store up to 13.2 petabytes (petabyte = 1000 TB) with a throughput rate of 55TB/hour. We defer the discussion of disk storage technology called RAID, and of storage area networks and network-attached storage, 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 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.

5. Called *interrecord gaps* in tape terminology.

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 read a continuous stream of blocks

Figure 13.3
Interleaved concurrency versus parallel execution.

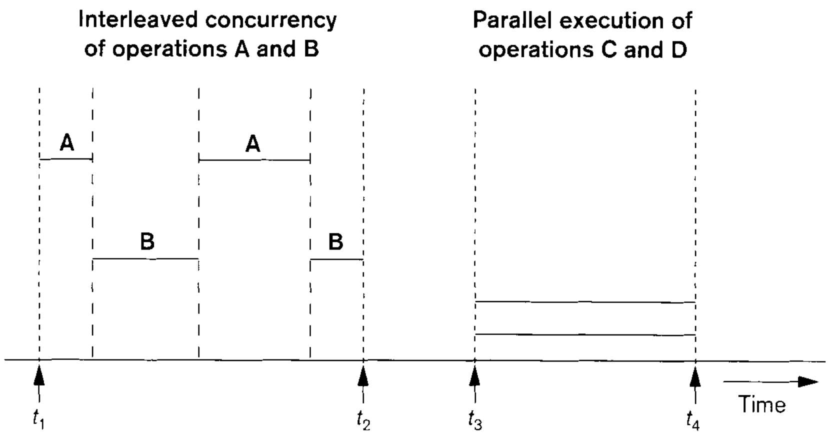
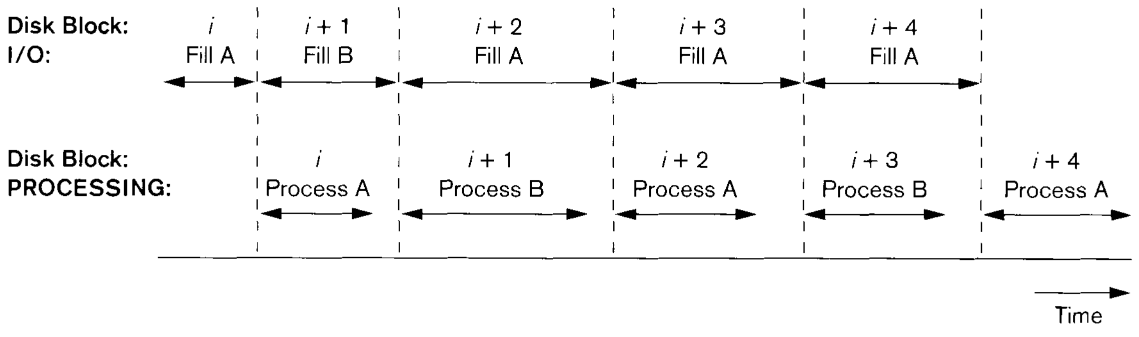

Interleaved concurrency of operations A and B
Parallel execution of operations C and D
A
A
B
B
t_1
t_2
t_3
t_4
Time

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



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

13.4 Placing File Records on Disk

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

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 corresponds to a particular **field** of the record. Records usually describe entities and their attributes. For example, an EMPLOYEE record represents an employee entity, and each field value in the record specifies some attribute of that employee, such as Name, Birth_date, Salary, or Supervisor. A collection of field names and their corresponding data types constitutes a **record type** or **record format** definition. A **data type**, associated with each field, specifies the types of values a field can take.

The data type of a field is usually one of the standard data types used in programming. These include numeric (integer, long integer, or floating point), string of characters (fixed-length or varying), Boolean (having 0 and 1 or TRUE and FALSE values only), and sometimes specially coded **date** and **time** data types. The number of bytes required for each data type is fixed for a given computer system. An integer may require 4 bytes, a long integer 8 bytes, a real number 4 bytes, a Boolean 1 byte, a date 10 bytes (assuming a format of YYYY-MM-DD), and a fixed-length string of *k* characters *k* bytes. Variable-length strings may require as many bytes as there are characters in each field value. For example, an EMPLOYEE record type may be defined—using the C programming language notation—as the following structure:

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

In 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 13.5(a) have a record size of 71 bytes. Every record has the same fields, and field lengths are fixed, so the system can identify the starting byte position of each field relative to the starting position of the record. This facilitates locating field values by programs that access such files. Notice that it is possible to represent a file that logically should have variable-length records as a fixed-length records file. For example, in the case of optional fields, we could have *every field* included in *every file record* but store a special NULL value if no value exists for that field. For a repeating field, we could allocate as many spaces in each record as the *maximum 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. Now we 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, as shown in Figure 13.5(b), or we can store the length in bytes of the field in the record, preceding the field value.

A file of records with *optional fields* can be formatted in different ways. If the total number of fields for the record type is large, but the number of fields that actually appear in a typical record is small, we can include in each record a sequence of

: same
: file is
ve dif-
y have

lds are
eld of

fields
lled a
eating

ds are
ecords

g size
e clus-
PORT
ENT's

bytes.
iden-
of the
Notice
cords
could
value
each
space
pro-
a file

know
cular
is ? or
ength
eld in

total
ually
ce of

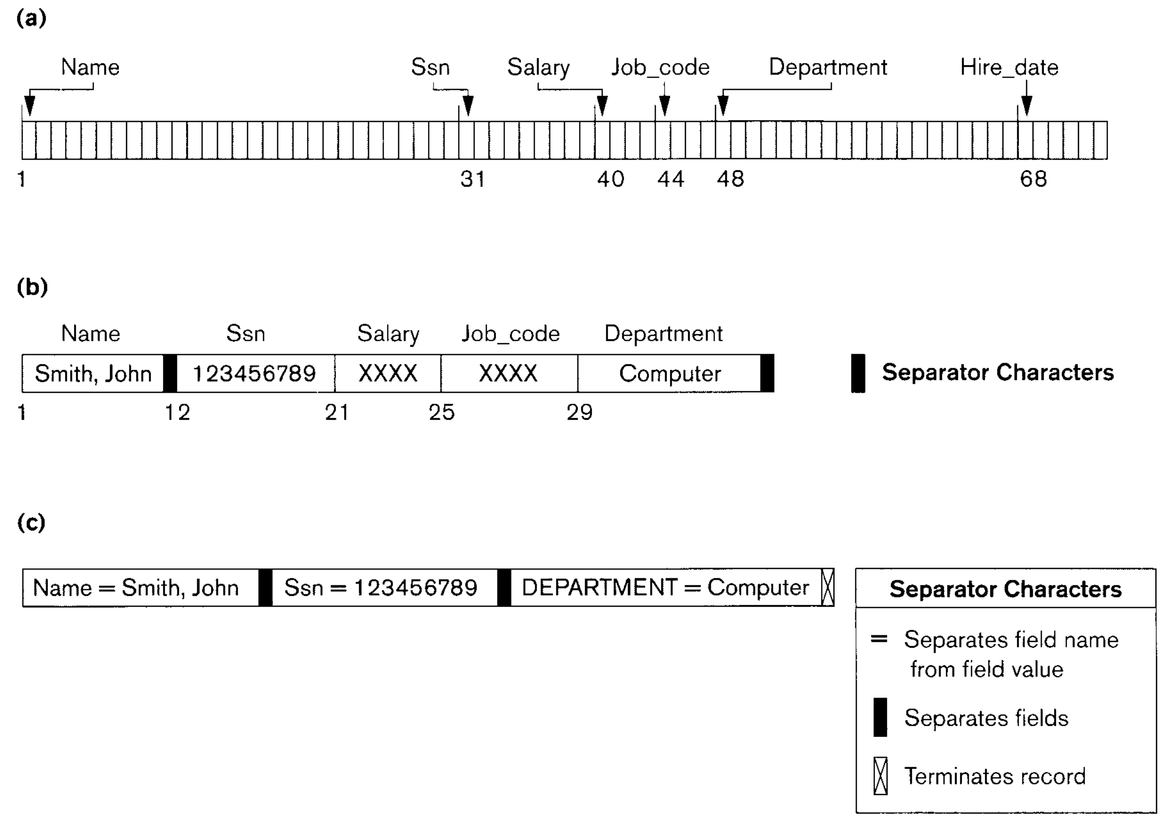


Figure 13.5
Three record storage formats. (a) A fixed-length record with six fields and size of 71 bytes. (b) A record with two variable-length fields and three fixed-length fields. (c) A variable-field record with three types of separator characters.

<field-name, field-value> pairs rather than just the field values. Three types of separator characters are used in Figure 13.7(c), although we could use the same separator character for the first two purposes—separating the field name from the field value and separating one field from the next field. A more practical option is to assign a short **field type** code—say, an integer number—to each field and include in each record a sequence of <field-type, field-value> pairs rather than <field-name, field-value> pairs.

A *repeating field* needs one separator character to separate the repeating values of the field and another separator character to indicate termination of the field. Finally, for a file that includes *records of different types,* each record is preceded by a **record type** indicator. Understandably, programs that process files of variable-length records—which are usually part of the file system and hence hidden from the typical programmers—need to be more complex than those for fixed-length records, where the starting position and size of each field are known and fixed.[6]

13.4.3 Record Blocking and Spanned versus Unspanned Records

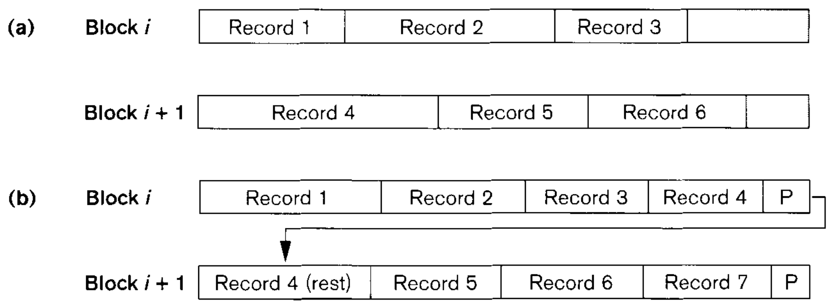
The records of a file must be allocated to disk blocks because a block is the *unit of data transfer* between disk and memory. When the block size is larger than the record size, each block will contain numerous records, although some files may have unusually large records that cannot fit in one block. 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

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

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*

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



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

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

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

When several file records satisfy a search condition, the *first* record—with respect to the physical sequence of file records—is initially located and designated the **current record**. Subsequent search operations commence from this record and locate the *next* record in the file that satisfies the condition.

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

- **Open.** Prepares the file for reading or writing. Allocates appropriate buffers (typically at least two) to hold file blocks from disk, and retrieves the file header. Sets the file pointer to the beginning of the file.
- **Reset.** Sets the file pointer of an open file to the beginning of the file.
- **Find (or Locate).** Searches for the first record that satisfies a search condition. Transfers the block containing that record into a main memory buffer (if it is not already there). The file pointer points to the record in the buffer and it becomes the *current record.* Sometimes, different verbs are used to indicate whether the located record is to be retrieved or updated.
- **Read (or Get).** Copies the current record from the buffer to a program variable in the user program. This command may also advance the current record pointer to the next record in the file, which may necessitate reading the next file block from disk.
- **FindNext.** Searches for the next record in the file that satisfies the search condition. Transfers the block containing that record into a main memory buffer (if it is not already there). The record is located in the buffer and becomes the current record.
- **Delete.** Deletes the current record and (eventually) updates the file on disk to reflect the deletion.
- **Modify.** Modifies some field values for the current record and (eventually) updates the file on disk to reflect the modification.

- **Insert.** Inserts a new record in the file by locating the block where the record is to be inserted, transferring that block into a main memory buffer (if it is not already there), writing the record into the buffer, and (eventually) writing the buffer to disk to reflect the insertion.
- **Close.** Completes the file access by releasing the buffers and performing any other needed cleanup operations.

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

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

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

- **FindAll.** Locates *all* the records in the file that satisfy a search condition.
- **Find (or Locate) *n*.** Searches for the first record that satisfies a search condition and then continues to locate the next $n - 1$ records satisfying the same condition. Transfers the blocks containing the n records to the main memory buffer (if not already there).
- **FindOrdered.** Retrieves all the records in the file in some specified order.
- **Reorganize.** Starts the reorganization process. As we shall see, some file organizations require periodic reorganization. An example is to reorder the file records by sorting them on a specified field.

At this point, it is worthwhile to note the difference between the terms *file organization* and *access method.* A **file organization** refers to the organization of the data of a file into records, blocks, and access structures; this includes the way records and blocks are placed on the storage medium and interlinked. An **access method**, on the other hand, provides a group of operations—such as those listed earlier—that can be applied to a file. In general, it is possible to apply several access methods to a file organization. Some access methods, though, can be applied only to files organized in certain ways. For example, we cannot apply an indexed access method to a file without an index (see Chapter 14).

Usually, we expect to use some search conditions more than others. Some files may be **static**, meaning that update operations are rarely performed; other, more **dynamic** files may change frequently, so update operations are constantly applied to them. A successful file organization should perform as efficiently as possible the operations we expect to *apply frequently* to the file. For example, consider the EMPLOYEE file, as shown in Figure 13.5(a), which stores the records for current employees in a company. We expect to insert records (when employees are hired), delete records (when employees leave the company), and modify records (for example, when an employee's salary or job is changed). Deleting or modifying a record requires a selection condition to identify a particular record or set of records. Retrieving one or more records also requires a selection condition.

If users expect mainly to apply a search condition based on Ssn, the designer must choose a file organization that facilitates locating a record given its Ssn value. This may involve physically ordering the records by Ssn value or defining an index on Ssn (see Chapter 14). Suppose that a second application uses the file to generate employees' paychecks and requires that paychecks are grouped by department. For this application, it is best to 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 14, we discuss methods for organizing records of a file on disk. Several general techniques, such as ordering, hashing, and indexing, are used to create access methods. Additionally, various general techniques for handling insertions and deletions work with many file organizations.

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 14. It is also used to collect and store data records for future use.

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

To delete a record, a program must first find its block, copy the block into a buffer, delete the record from the buffer, and finally **rewrite the block** back to the disk. This leaves unused space in the disk block. Deleting a large number of records in this way

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

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).

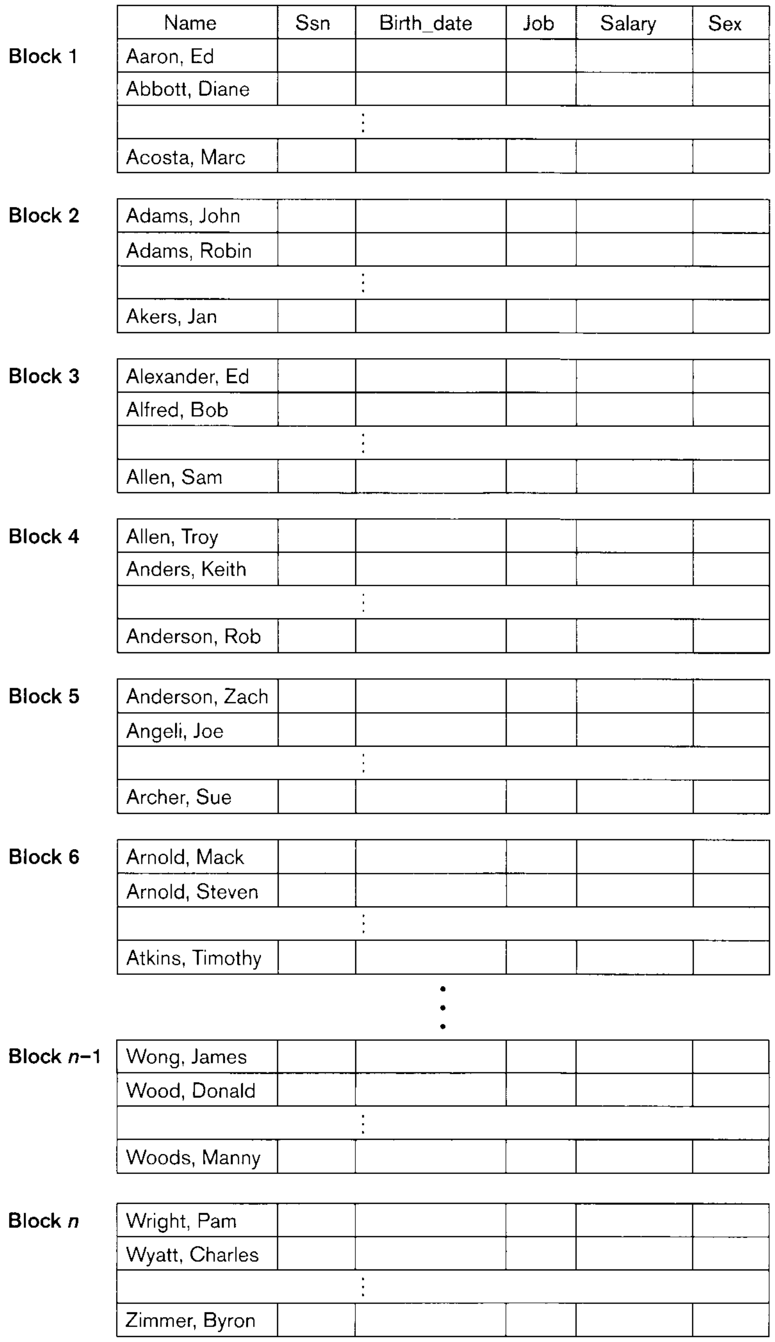
For a file of unordered *fixed-length records* using *unspanned blocks* and *contiguous allocation,* it is straightforward to access any record by its **position** in the file. If the file records are numbered 0, 1, 2, . . . , $r - 1$ and the records in each block are numbered 0, 1, . . . , $bfr - 1$, where bfr is the blocking factor, then the ith record of the file is located in block $\lfloor (i/bfr) \rfloor$ and is the $(i \bmod bfr)$th record in that block. Such a file is often called a **relative** or **direct file** because records can easily be accessed directly by their relative positions. Accessing a record by its position does not help locate a record based on a search condition; however, it facilitates the construction of access paths on the file, such as the indexes discussed in Chapter 14.

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

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



	Name	Ssn	Birth_date	Job	Salary	Sex
Block 1	Aaron, Ed					
	Abbott, Diane					
			⋮			
	Acosta, Marc					
Block 2	Adams, John					
	Adams, Robin					
			⋮			
	Akers, Jan					
Block 3	Alexander, Ed					
	Alfred, Bob					
			⋮			
	Allen, Sam					
Block 4	Allen, Troy					
	Anders, Keith					
			⋮			
	Anderson, Rob					
Block 5	Anderson, Zach					
	Angeli, Joe					
			⋮			
	Archer, Sue					
Block 6	Arnold, Mack					
	Arnold, Steven					
			⋮			
	Atkins, Timothy					
				⋮		
Block ***n*****−1**	Wong, James					
	Wood, Donald					
			⋮			
	Woods, Manny					
Block ***n***	Wright, Pam					
	Wyatt, Charles					
			⋮			
	Zimmer, Byron					

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

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 $1, 2, \ldots, b$; 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

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

A search criterion involving the conditions $>$, $<$, $\geq$, and $\leq$, on the ordering field is quite efficient, since the physical ordering of records means that all records satisfying the condition are contiguous in the file. For example, referring to Figure 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: the search condition to locate the record and the field to be modified. If the search condition involves the ordering key field, we can locate the record using a binary search; otherwise we must do a linear search. A nonordering field can be modified by changing the record and rewriting it in the same physical location on disk—assuming fixed-length records. Modifying the ordering field means that the record can change its position in the file. This requires deletion of the old record followed by insertion of the modified record.

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

Table 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. If the ordering attribute is not a key, the file is called a **clustered file**.

13.8 Hashing Techniques

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

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

Table 13.2
Average Access Times for a File of *b* Blocks under Basic File Organizations

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

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 13.8.1; then we show how it is modified to store external files on disk in Section 13.8.2. In Section 13.8.3 we discuss techniques for extending hashing to dynamically growing files.

13.8.1 Internal Hashing

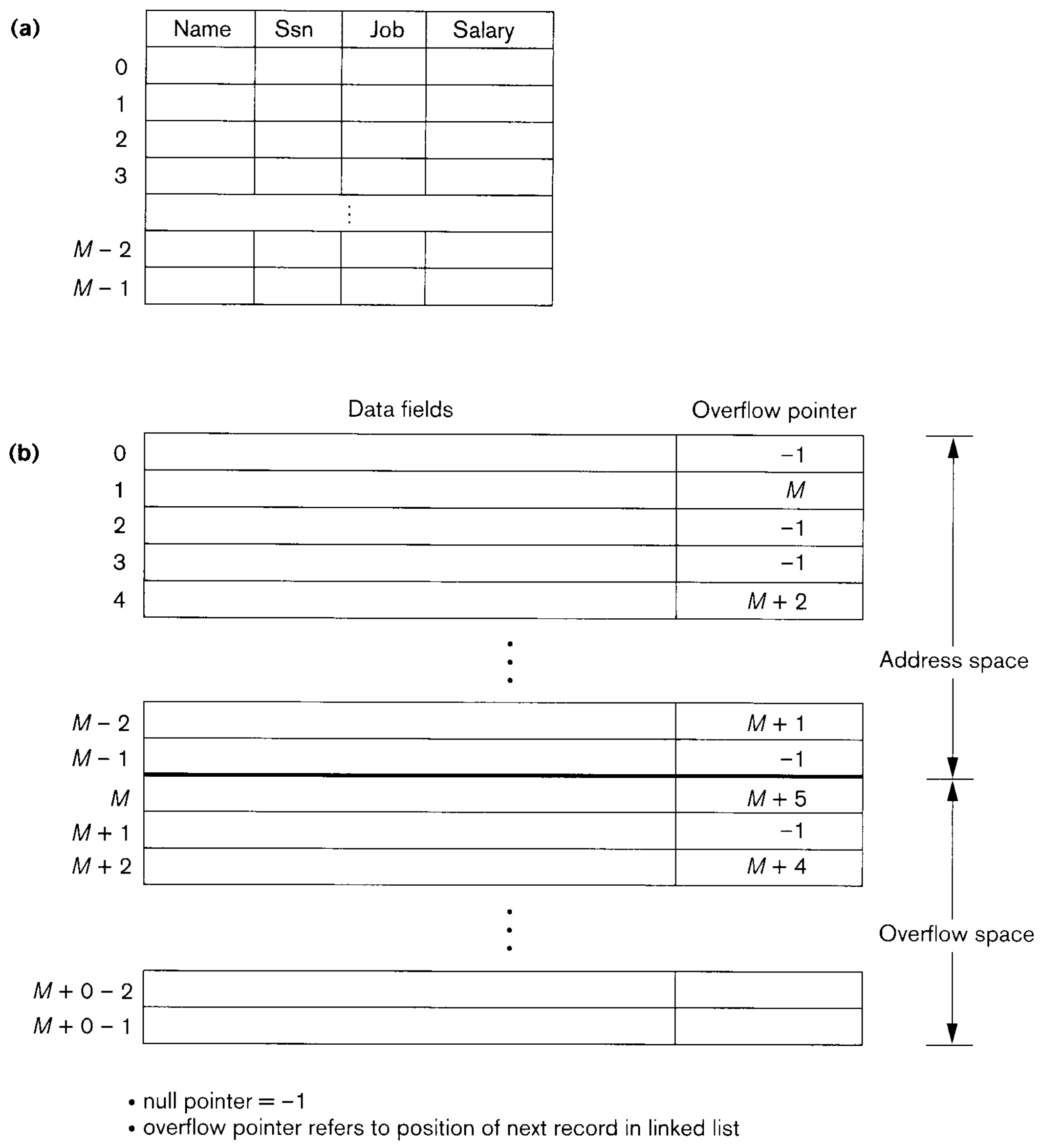
For internal files, hashing is typically implemented as a **hash table** through the use of an array of records. Suppose that the array index range is from 0 to $M - 1$, as shown in Figure 13.8(a); then we have M **slots** whose addresses correspond to the array indexes. We choose a hash function that transforms the hash field value into an integer between 0 and $M - 1$. One common hash function is the $h(K) = K$ **mod** M function, which returns the remainder of an integer hash field value K after division by M; this value is then used for the record address.

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 13.2(a) can be used to calculate the hash address. We assume that the code function returns the numeric code of a character and that we are given a hash field value K of type *K: array* [1..20] *of char* (in PASCAL) or *char* K[20] (in C).

Algorithm 13.2. Two simple hashing algorithms. (a) Applying the mod hash function to a character string K. (b) Collision resolution by open addressing.

```
(a)  $temp \leftarrow 1$;
     for $i \leftarrow 1$ to 20 do $temp \leftarrow temp * \text{code}(K[i]) \bmod M$ ;
     $hash\_address \leftarrow temp \bmod M$;
(b)  $i \leftarrow hash\_address(K)$; $a \leftarrow i$;
     if location $i$ is occupied
        then begin $i \leftarrow (i + 1) \bmod M$;
               while $(i \neq a)$ and location $i$ is occupied
                  do $i \leftarrow (i + 1) \bmod M$;
        if ($i = a$) then all positions are full
        else $new\_hash\_address \leftarrow i$;
    end;
```

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



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 13.2(b) may be used for this purpose.
- **Chaining.** For this method, various overflow locations are kept, usually by extending the array with a number of overflow positions. Additionally, a pointer field is added to each record location. A collision is resolved by placing the new record in an unused overflow location and setting the pointer of the occupied hash address location to the address of that overflow location. A linked list of overflow records for each hash address is thus maintained, as shown in Figure 13.8(b).
- **Multiple hashing.** The program applies a second hash function if the first results in a collision. If another collision results, the program uses open addressing or applies a third hash function and then uses open addressing if necessary.

Each collision resolution method requires its own algorithms for insertion, retrieval, and deletion of records. The algorithms for chaining are the simplest. Deletion algorithms for open addressing are rather tricky. Data structures textbooks discuss internal hashing algorithms in more detail.

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

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

address space when the mod hashing function is used. Other hash functions may require M to be a power of 2.

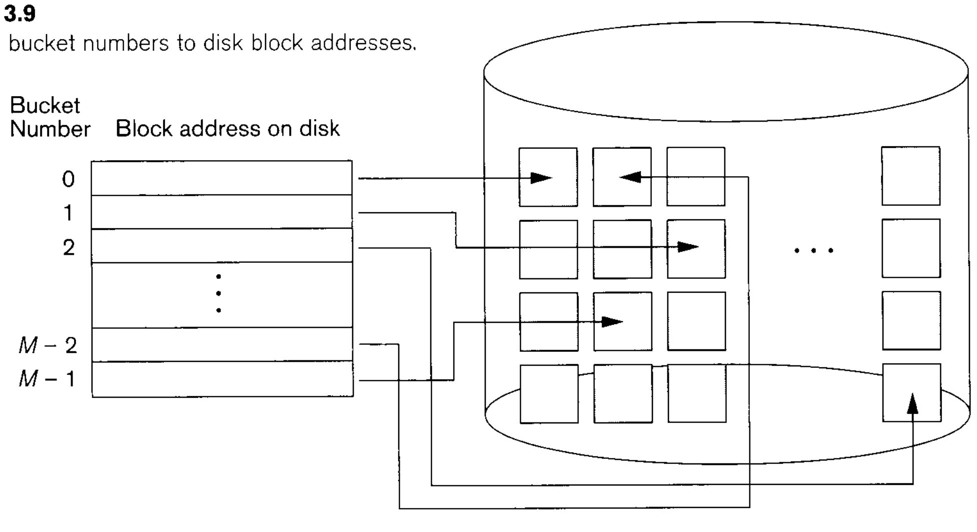
13.8.2 External Hashing for Disk Files

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 assigning an absolute block address to the bucket. A table maintained in the file header converts the bucket number into the corresponding disk block address, as illustrated in Figure 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 13.10. 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

Figure 13.9
Matching bucket numbers to disk block addresses.



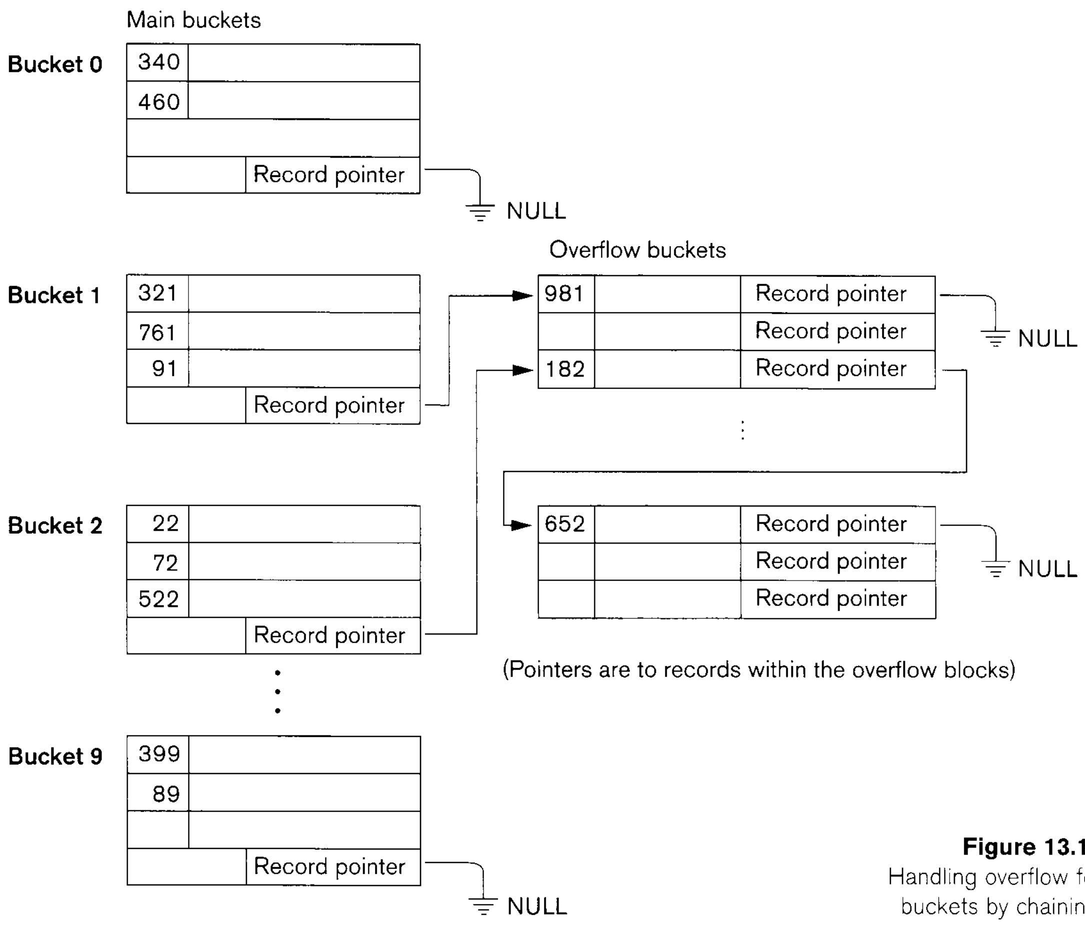


Figure 13.10
Handling overflow for buckets by chaining.

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

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 14). 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 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 $d = 3$.

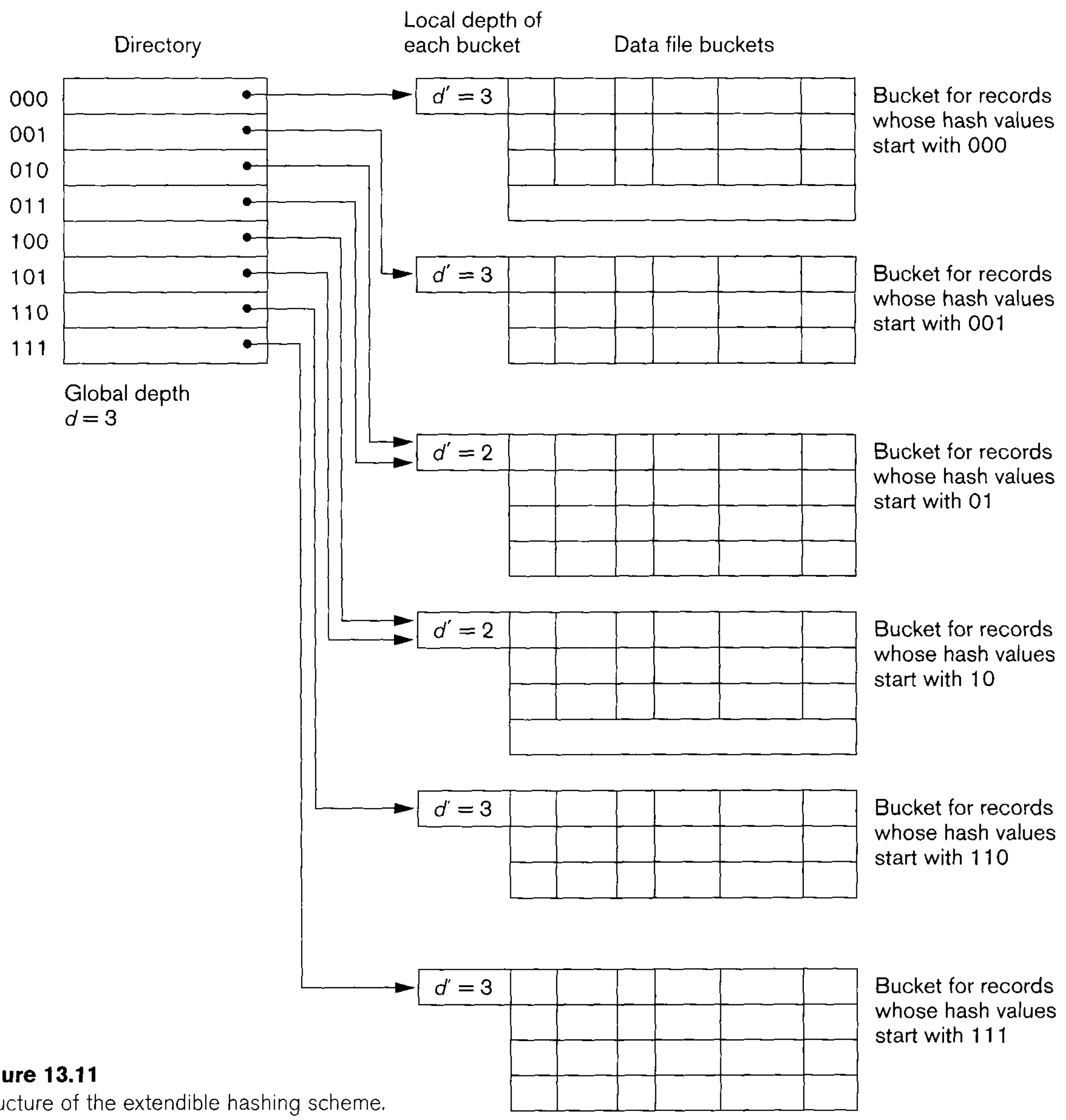
The value of d can be increased or decreased by one at a time, thus doubling or halving the number of entries in the directory array. Doubling is needed if a bucket, whose local depth d' is equal to the global depth d, overflows. Halving occurs if $d > d'$ for all the buckets after some deletions occur. 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. Additionally, no space is allocated in extendible hashing for future growth, but additional buckets can be allocated dynamically as needed. The space overhead for the directory table is negligible. The maximum directory size is 2^k, where k is the number of bits in the hash value. Another advantage is that splitting causes minor reorganization in most cases, since only the records in one bucket are redistributed to the two new buckets. The only time reorganization is more expensive is when the directory has to be doubled (or halved). A disadvantage is that the directory must be searched before accessing the buckets themselves, resulting in two block accesses instead of one in static hashing. This performance penalty is considered minor and 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 $0, 1, \ldots, M - 1$ and uses the mod hash



Directory
Local depth of each bucket
Data file buckets
000
001
010
011
100
101
110
111
Global depth
$d = 3$
$d' = 3$
Bucket for records whose hash values start with 000
$d' = 3$
Bucket for records whose hash values start with 001
$d' = 2$
Bucket for records whose hash values start with 01
$d' = 2$
Bucket for records whose hash values start with 10
$d' = 3$
Bucket for records whose hash values start with 110
$d' = 3$
Bucket for records whose hash values start with 111

Figure 13.11
Structure of the extendible hashing scheme.

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 $h_{i+j}(K) = K \bmod (2^j M)$ is used, where $j = 0, 1, 2, \ldots$; a new hashing function h_{i+j+1} is needed whenever all the buckets $0, 1, \ldots, (2^j M) - 1$ have been split and n is reset to 0. The search for a record with hash key value K is given by Algorithm 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 factor of the file falls below a certain threshold. Blocks are combined linearly, and N is decremented appropriately. The file load can be used to trigger both splits and combinations; in this manner the file load can be kept within a desired range. Splits can be triggered when the load exceeds a certain threshold—say, 0.9—and combinations can be triggered when the load falls below another threshold—say, 0.7.

Algorithm 13.3. The Search Procedure for Linear Hashing

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

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

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

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

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 struc-

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.

ture 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 with processor technology in performance and reliability.

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

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.

Table 13.3
Trends in Disk Technology

	1993 Parameter Values*	Historical Rate of Improvement per Year (%)*	2003 Values**
Areal density	50–150 Mb/sq. inch	27	36 Gb/sq. inch
Linear density	40,000–60,000 bits/inch	13	570 Kb/inch
Inter-track density	1500–3000 tracks/inch	10	64,000 tracks/inch
Capacity (3.5-inch 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 ms

**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.

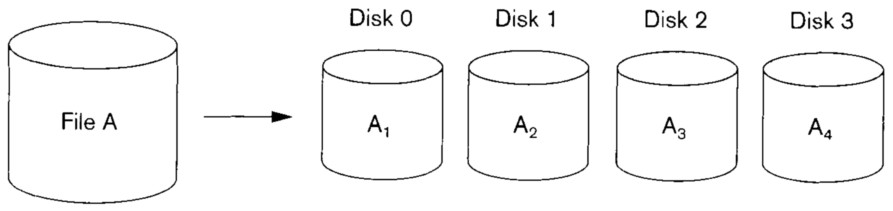


Figure 13.12
Data striping. File A is striped across four disks.

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: selecting a technique for computing the redundant information, and selecting a method of distributing the redundant information across the disk array. The first problem is addressed by using error correcting codes involving parity bits, or specialized codes such as Hamming codes. Under the parity scheme, a redundant disk may be considered as having the sum of all the data in the other disks. When a disk fails, the missing information can be constructed by a process similar to subtraction.

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

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 to 8192 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 jth 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 1/100th the reliability of a single disk. Thus, redundancy via error-correcting codes and disk mirroring is necessary to provide reliability along with high performance.

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

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 New Storage Systems

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

13.11.1 Storage Area Networks

With the rapid growth of electronic commerce, Enterprise Resource Planning (ERP) systems that integrate application data across organizations, and data warehouses that keep historical aggregate information (see Chapter 27), the demand for

of the
› com-
RAID

t write
›erfor-
·r, per-
k with
·edun-
ıpping
ındant
—four
hough

ıre out
5 dis-
ıpplies
›rotect
RAID

.ire the
critical
›r large
use of
g) and
pplica-
: num-
k-level
to I/O
ests for
ned.

ıat are
ıres.

ınning
ı ware-
ınd for

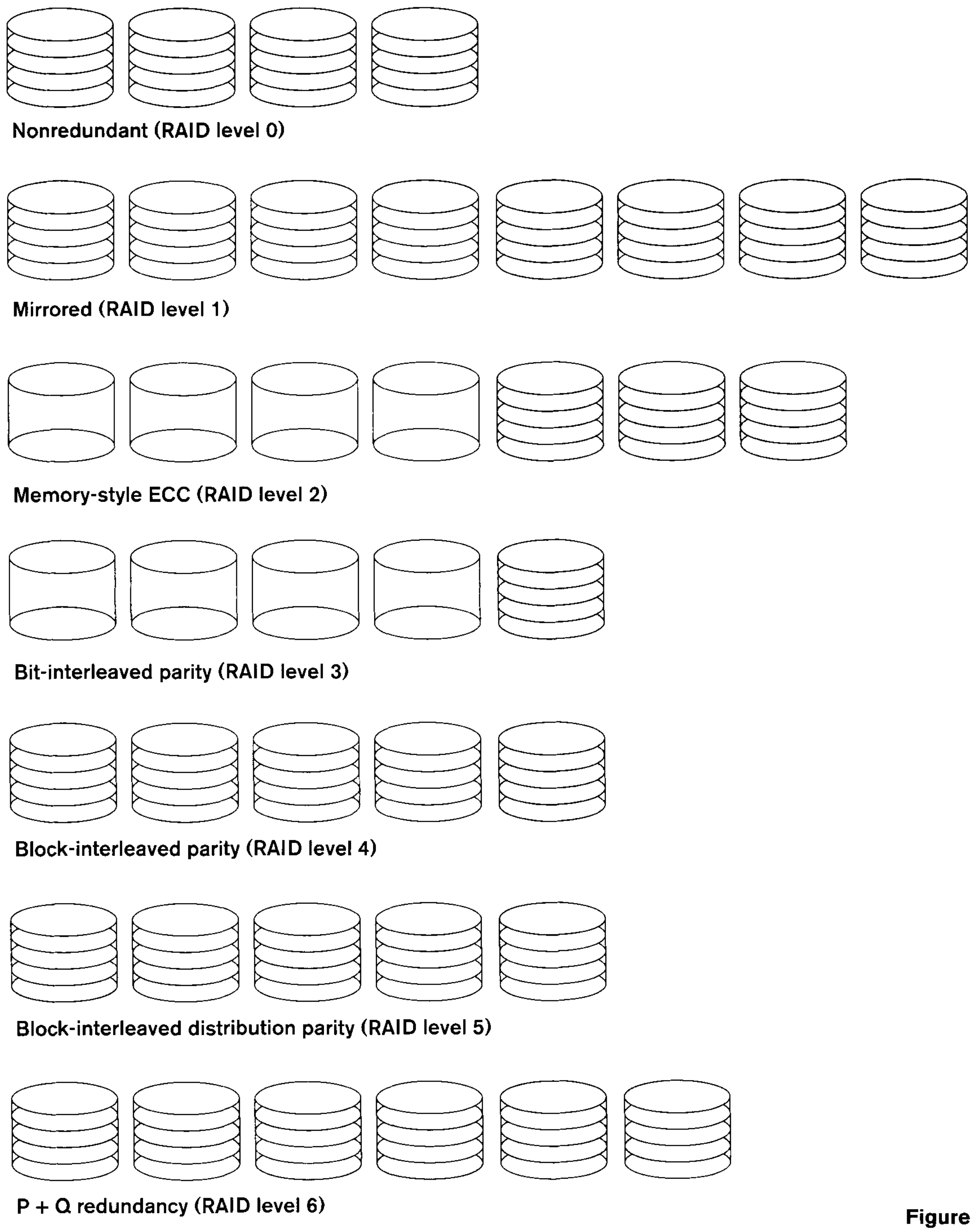

Nonredundant (RAID level 0)
Mirrored (RAID level 1)
Memory-style ECC (RAID level 2)
Bit-interleaved parity (RAID level 3)
Block-interleaved parity (RAID level 4)
Block-interleaved distribution parity (RAID level 5)
P + Q redundancy (RAID level 6)

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 permission.

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

13.11.2 Network-Attached Storage

With the phenomenal growth in digital data, particularly generated from multimedia and other enterprise applications, the need for high performance storage solutions at low cost has become extremely important. **Network-Attached Storage** (NAS) devices are among the latest of storage devices being used for this purpose. These devices are, in fact, servers that do not provide any of the common server services, but simply allow the addition of storage for file sharing. NAS devices allow vast amounts of hard disk storage space to be added to a network and can make that space available to multiple servers without shutting them down for maintenance and upgrades. NAS devices can reside anywhere on a Local Area Network (LAN) and may be combined in different configurations. A single hardware device, often called the **NAS box** or **NAS head**, acts as the interface between the NAS system and network clients. These NAS devices require no monitor, keyboard or mouse. One or more disk or tape drives can be attached to many NAS systems to increase total capacity. Clients connect to the NAS head rather than to the individual storage devices. A NAS can store any data that appears in the form of files, such as email boxes, Web content, remote system backups, and so on. In that sense, NAS devices are being deployed as a replacement for traditional file servers.

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

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. To reduce the average block access time, double buffering can be used when accessing consecutive disk blocks. Other disk parameters are discussed in Appendix B. We presented different ways of storing file records on disk. File records are grouped into disk blocks and can be fixed length or variable length, spanned or unspanned, and of the same record type or mixed types. We discussed the file header, which describes the record formats and keeps track of the disk addresses of the file blocks. Information in the file header is used by system software accessing the file records.

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

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

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

Hashing provides very fast access to an arbitrary record of a file, given the value of its hash key. The most suitable method for external hashing is the bucket technique, with one or more contiguous blocks corresponding to each bucket. Collisions causing bucket overflow are handled by chaining. Access on any nonhash field is slow, and so is ordered access of the records on any field. We discussed two hashing techniques for files that grow and shrink in the number of records dynamically: 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. How does double buffering improve block access time?

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

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