

## Disk Storage, Basic File Structures, Hashing, and Modern Storage Architectures

Databases are stored physically as files of records, which are typically stored on magnetic disks. This chapter and the next deal with the organization of databases in storage and the techniques for accessing them efficiently using various algorithms, some of which require auxiliary data structures called *indexes*. These structures are often referred to as **physical database file structures** and are at the physical level of the three-schema architecture described in Chapter 2. We start in Section 16.1 by introducing the concepts of computer storage hierarchies and how they are used in database systems. Section 16.2 is devoted to a description of magnetic disk storage devices and their characteristics, flash memory, and solid-state drives and optical drives and magnetic tape storage devices used for archiving data. We also discuss techniques for making access from disks more efficient. After discussing different storage technologies, we turn our attention to the methods for physically organizing data on disks. Section 16.3 covers the technique of double buffering, which is used to speed retrieval of multiple disk blocks. We also discuss buffer management and buffer replacement strategies. In Section 16.4 we discuss various ways of formatting and storing file records on disk. Section 16.5 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 16.6; ordered records, in Section 16.7; and hashed records, in Section 16.8.

Section 16.9 briefly introduces files of mixed records and other primary methods for organizing records, such as B-trees. These are particularly relevant for storage of object-oriented databases, which we discussed in Chapter 11. Section 16.10

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 16.11 we describe modern developments in the storage architectures that are important for storing enterprise data: storage area networks (SANs), network-attached storage (NAS), iSCSI (Internet SCSI—small computer system interface), and other network-based storage protocols, which make storage area networks more affordable without the use of the Fibre Channel infrastructure and hence are becoming widely accepted in industry. We also discuss storage tiering and object-based storage. Section 16.12 summarizes the chapter. In Chapter 17 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 16 and 17 may be browsed through or even omitted by readers who have already studied file organizations and indexing in a separate course. The material covered here, in particular Sections 16.1 through 16.8, is necessary for understanding Chapters 18 and 19, which deal with query processing and optimization, as well as database tuning for improving performance of queries.

## 16.1 Introduction

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

- **Primary storage.** This category includes storage media that can be operated on directly by the computer's *central processing unit* (CPU), such as the computer's main memory and smaller but faster cache memories. Primary storage usually provides fast access to data but is of limited storage capacity. Although main memory capacities have been growing rapidly in recent years, they are still more expensive and have less storage capacity than demanded by typical enterprise-level databases. The contents of main memory are lost in case of power failure or a system crash.
- **Secondary storage.** The primary choice of storage medium for online storage of enterprise databases has been magnetic disks. However, flash memories are becoming a common medium of choice for storing moderate amounts of permanent data. When used as a substitute for a disk drive, such memory is called a **solid-state drive** (SSD).
- **Tertiary storage.** Optical disks (CD-ROMs, DVDs, and other similar storage media) and tapes are removable media used in today's systems as offline storage for archiving databases and hence come under the category called tertiary storage. These devices usually have a larger capacity, cost less, and provide slower access to data than do primary storage devices. Data in secondary or tertiary storage cannot be processed directly by the CPU; first it must be copied into primary storage and then processed by the CPU.

We first give an overview of the various storage devices used for primary, secondary, and tertiary storage in Section 16.1.1, and in Section 16.1.2 we discuss how databases are typically handled in the storage hierarchy.

### 16.1.1 Memory Hierarchies and Storage Devices<sup>1</sup>

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

At the *primary storage level*, the memory hierarchy includes, at the most expensive end, **cache memory**, which is a static RAM (random access memory). Cache memory is typically used by the CPU to speed up execution of program instructions using techniques such as prefetching and pipelining. The next level of primary storage is DRAM (dynamic RAM), which provides the main work area for the CPU for keeping program instructions and data. It is popularly called **main memory**. The advantage of DRAM is its low cost, which continues to decrease; the drawback is its volatility<sup>2</sup> and lower speed compared with static RAM.

At the *secondary and tertiary storage level*, the hierarchy includes magnetic disks; **mass storage** in the form of CD-ROM (compact disk–read-only memory) and DVD (digital video disk or digital versatile disk) devices; and finally tapes at the least expensive end of the hierarchy. The **storage capacity** is measured in kilobytes (Kbyte or 1,000 bytes), megabytes (MB or 1 million bytes), gigabytes (GB or 1 billion bytes), and even terabytes (1,000 GB). The word *petabyte* (1,000 terabytes or  $10^{15}$  bytes) is now becoming relevant in the context of very large repositories of data in physics, astronomy, earth sciences, and other scientific applications.

Programs reside and execute in dynamic random-access memory (DRAM). Generally, large permanent databases reside on secondary storage (magnetic disks), and portions of the database are read into and written from buffers in main memory as needed. Nowadays, personal computers and workstations have large main memories of hundreds of megabytes of RAM and DRAM, so it is becoming possible to load a large part of the database into main memory. Eight to sixteen GB of main memory is becoming commonplace on laptops, and servers with 256 GB capacity are not uncommon. In some cases, entire databases can be kept in main memory (with a backup copy on magnetic disk), which results in **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.

**Flash Memory.** Between DRAM and magnetic disk storage, another form of memory, **flash memory**, is becoming common, particularly because it is nonvolatile.

---

<sup>1</sup>The authors appreciate the valuable input of Dan Forsyth regarding the current status of storage systems in enterprises. The authors also wish to thank Satish Damle for his suggestions.

<sup>2</sup>Volatile memory typically loses its contents in case of a power outage, whereas nonvolatile memory does not.

Flash memories are high-density, high-performance memories using EEPROM (electrically erasable programmable read-only memory) technology. The advantage of flash memory is the fast access speed; the disadvantage is that an entire block must be erased and written over simultaneously. Flash memories come in two types called NAND and NOR flash based on the type of logic circuits used. The NAND flash devices have a higher storage capacity for a given cost and are used as the data storage medium in appliances with capacities ranging from 8 GB to 64 GB for the popular cards that cost less than a dollar per GB. Flash devices are used in cameras, MP3/MP4 players, cell phones, PDAs (personal digital assistants), and so on. USB (universal serial bus) flash drives or USB sticks have become the most portable medium for carrying data between personal computers; they have a flash memory storage device integrated with a USB interface.

**Optical Drives.** The most popular form of optical removable storage is CDs (compact disks) and DVDs. CDs have a 700-MB capacity whereas DVDs have capacities ranging from 4.5 to 15 GB. CD-ROM(compact disk – read only memory) disks store data optically and are read by a laser. CD-ROMs contain prerecorded data that cannot be overwritten. The version of compact and digital video disks called CD-R (compact disk recordable) and DVD-R or DVD+R, which are also known as 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.<sup>3</sup> A higher capacity format for DVDs called Blu-ray DVD can store 27 GB per layer, or 54 GB in a two-layer disk. **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 **tertiary storage** is continuing to decline because of the rapid decrease in cost and the increase in capacities of magnetic disks. Most personal computer disk drives now read CD-ROM and DVD disks. Typically, drives are CD-R (compact disk recordable) that can create CD-ROMs and audio CDs, as well as record on DVDs.

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

Many large organizations are using terabyte-sized databases. The term **very large database** can no longer be precisely defined because disk storage capacities are on

---

<sup>3</sup>Their rotational speeds are lower (around 400 rpm), giving higher latency delays and low transfer rates (around 100 to 200 KB/second) for a 1X drive.  $nX$  drives (e.g., 16X ( $n = 16$ )) are supposed to give  $n$  times higher transfer rate by multiplying the rpm  $n$  times. The 1X DVD transfer rate is about 1.385 MB/s.

**Table 16.1** Types of Storage with Capacity, Access Time, Max Bandwidth (Transfer Speed), and Commodity Cost

Type	Capacity*	Access Time	Max Bandwidth	Commodity Prices (2014)**
Main Memory- RAM	4GB–1TB	30ns	35GB/sec	\$100–\$20K
Flash Memory- SSD	64 GB–1TB	50μs	750MB/sec	\$50–\$600
Flash Memory- USB stick	4GB–512GB	100μs	50MB/sec	\$2–\$200
Magnetic Disk	400 GB–8TB	10ms	200MB/sec	\$70–\$500
Optical Storage	50GB–100GB	180ms	72MB/sec	\$100
Magnetic Tape	2.5TB–8.5TB	10s–80s	40–250MB/sec	\$2.5K–\$30K
Tape jukebox	25TB–2,100,000TB	10s–80s	250MB/sec–1.2PB/sec	\$3K–\$1M+

\*Capacities are based on commercially available popular units in 2014.

\*\*Costs are based on commodity online marketplaces.

the rise and costs are declining. Soon the term *very large database* may be reserved for databases containing hundreds of terabytes or petabytes.

To summarize, a hierarchy of storage devices and storage systems is available today for storage of data. Depending upon the intended use and application requirements, data is kept in one or more levels of this hierarchy. Table 16.1 summarizes the current state of these devices and systems and shows the range of capacities, average access times, bandwidths (transfer speeds), and costs on the open commodity market. Cost of storage is generally going down at all levels of this hierarchy.

### 16.1.2 Storage Organization of Databases

Databases typically store large amounts of data that must persist over long periods of time, and hence the data is often referred to as **persistent data**. Parts of this data are accessed and processed repeatedly during the storage period. This contrasts with the notion of **transient data**, which persists 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.<sup>4</sup>
- 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.

<sup>4</sup>This statement is being challenged by recent developments in main memory database systems. Examples of prominent commercial systems include HANA by SAP and Timesten by Oracle.

Some of the newer technologies—such as solid-state drive (SSD) disks—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 16.1.1. The levels may range from the highest speed main memory level storage to the tape jukebox low speed offline storage. 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 much less than storage on disk. With some intervention by an operator—or an automatic loading device—tapes or optical removable disks must be loaded and read before the data becomes available for processing. 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. We will discuss some of the techniques for making disk access more efficient in Section 17.2.2.

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 16.6 through 16.9. 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 17.

## 16.2 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.

### 16.2.1 Hardware Description of Disk Devices

Magnetic disks are used for storing large amounts of data. The device that holds the disks is referred to as a **hard disk drive**, or **HDD**. The most basic unit of data on the disk is a single **bit** of information. By magnetizing an area on a disk in certain ways, one can make that area 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; 8 bits is the most common. 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 were used with laptops and desktops for many years—they contained a single disk typically holding from 400 KB to 1.5 MB; they are almost completely out of circulation. Hard disks for personal computers currently hold from several hundred gigabytes up to a few terabytes; and large disk packs used with servers and mainframes have capacities of hundreds of gigabytes. Disk capacities continue to grow as technology improves.

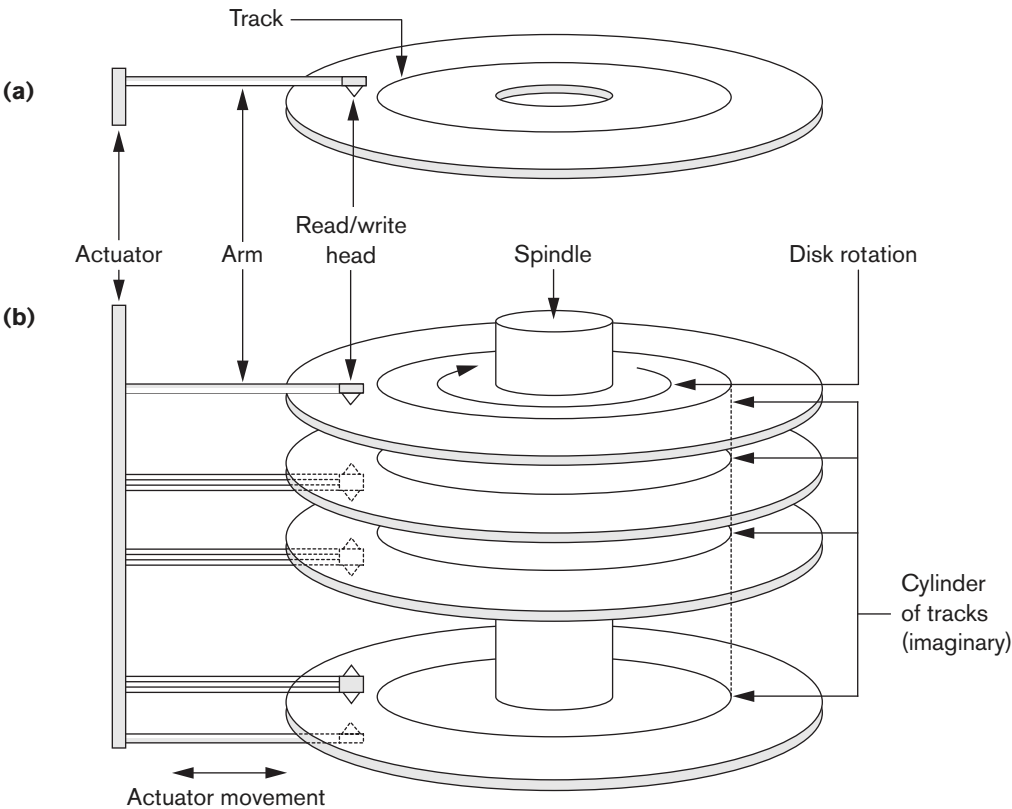
Whatever their capacity, all disks are made of magnetic material shaped as a thin circular disk, as shown in Figure 16.1(a), and protected by a plastic or acrylic cover. A disk is **single-sided** if it stores information on one of its surfaces only and **double-sided** if both surfaces are used. To increase storage capacity, disks are assembled into a **disk pack**, as shown in Figure 16.1(b), which may include many disks and therefore many surfaces. The two most common form factors are 3.5 and 2.5 inch diameter. Information is stored on a disk surface in concentric circles of *small width*,<sup>5</sup> 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 thousand to 152,000 on the disk drives shown in Table 16.2, and the capacity of each track typically ranges from tens of kilobytes 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 16.2(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

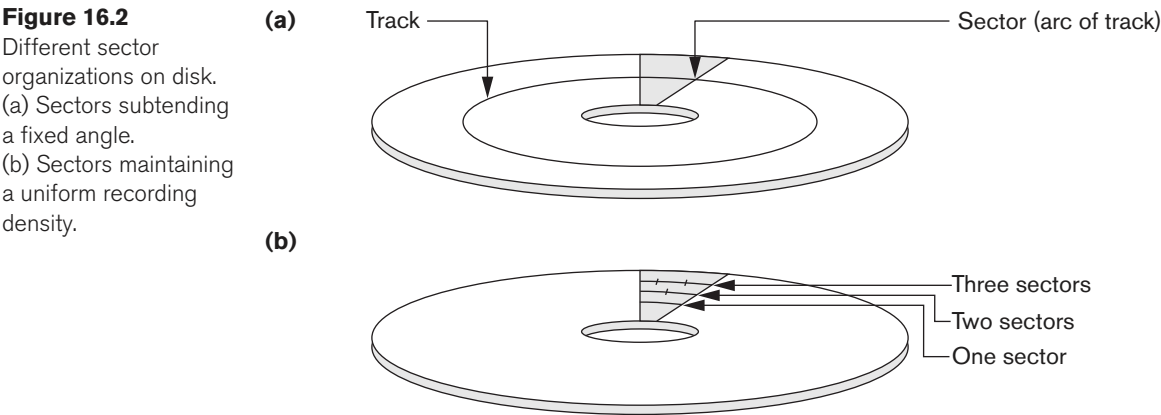
---

<sup>5</sup>In some disks, the circles are now connected into a kind of continuous spiral.





**Figure 16.1**  
(a) A single-sided disk with read/write hardware. (b) A disk pack with read/write hardware.



**Figure 16.2**  
Different sector organizations on disk.  
(a) Sectors subtending a fixed angle.  
(b) Sectors maintaining a uniform recording density.



away, thus maintaining a uniform density of recording, as shown in Figure 16.2(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. A common sector size is 512 bytes. 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

**Table 16.2** Specifications of Typical High-End Enterprise Disks from Seagate (a) Seagate Enterprise Performance 10 K HDD - 1200 GB

<b>Specifications</b>	<b>1200GB</b>
SED Model Number	ST1200MM0017
SED FIPS 140-2 Model Number	ST1200MM0027
Model Name	Enterprise Performance 10K HDD v7
Interface	6Gb/s SAS
<b>Capacity</b>	
Formatted 512 Bytes/Sector (GB)	1200
External Transfer Rate (MB/s)	600
<b>Performance</b>	
Spindle Speed (RPM)	10K
Average Latency (ms)	2.9
Sustained Transfer Rate Outer to Inner Diameter (MB/s)	204 to 125
Cache, Multisegmented (MB)	64
<b>Configuration/Reliability</b>	
Disks	4
Heads	8
Nonrecoverable Read Errors per Bits Read	1 per 10E16
Annualized Failure Rate (AFR)	0.44%
<b>Physical</b>	
Height (in/mm, max)	0.591/15.00
Width (in/mm, max)	2.760/70.10
Depth (in/mm, max)	3.955/100.45
Weight (lb/kg)	0.450/0.204

Courtesy Seagate Technology

(Continued)

**Table 16.2** (b) Internal Drive Characteristics of 300 GB–900 GB Seagate Drives

	ST900MM0006	ST600MM0006	ST450MM0006	ST300MM0006	
	ST900MM0026	ST600MM0026	ST450MM0026	ST300MM0026	
	ST900MM0046	ST600MM0046	ST450MM0046	ST300MM0046	
	ST900MM0036				
Drive capacity	900	600	450	300	GB (formatted, rounded off value)
Read/write data heads	6	4	3	2	
Bytes per track	997.9	997.9	997.9	997.9	KBytes (avg, rounded off values)
Bytes per surface	151,674	151,674	151,674	151,674	MB (unformatted, rounded off value)
Tracks per surface (total)	152	152	152	152	KTracks (user accessible)
Tracks per inch	279	279	279	279	KTPI (average)
Peak bits per inch	1925	1925	1925	1925	KBPI
Areal density	538	538	538	538	Gb/in <sup>2</sup>
Disk rotation speed	10K	10K	10K	10K	rpm
Avg rotational latency	2.9	2.9	2.9	2.9	ms

from 512 to 8192 bytes. A disk with hard-coded sectors often has the sectors subdivided or combined 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 16.2 illustrates the specifications of typical disks used on large servers in industry. The 10K prefix on disk names refers to the rotational speeds in rpm (revolutions per minute).

There is continuous improvement in the storage capacity and transfer rates associated with disks; they are also progressively getting cheaper—currently costing only a fraction of a dollar per megabyte of disk storage. Costs are going down so rapidly that costs as low as \$100/TB are already on the market.

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

reserved area in main storage that holds one disk block—is also provided. For a **read** command, the disk block is copied into the buffer; whereas for a **write** command, the contents of the buffer are copied into the disk block. Sometimes several contiguous blocks, called a **cluster**, may be transferred as a unit. In this case, the buffer size is adjusted to match the number of bytes in the cluster.

The actual hardware mechanism that reads or writes a block is the disk **read/write head**, which is part of a system called a **disk drive**. A disk or disk pack is mounted in the disk drive, which includes a motor that rotates the disks. A read/write head includes an electronic component attached to a **mechanical arm**. Disk packs with multiple surfaces are controlled by several read/write heads—one for each surface, as shown in Figure 16.1(b). All arms are connected to an **actuator** attached to another electrical motor, which moves the read/write heads in unison and positions them precisely over the cylinder of tracks specified in a block address.

Disk drives for hard disks rotate the disk pack continuously at a constant speed (typically ranging between 5,400 and 15,000 rpm). Once the read/write head is positioned on the right track and the block specified in the block address moves under the read/write head, the electronic component of the read/write head is activated to transfer the data. Some disk units have fixed read/write heads, with as many heads as there are tracks. These are called **fixed-head** disks, whereas disk units with an actuator are called **movable-head disks**. For fixed-head disks, a track or cylinder is selected by electronically switching to the appropriate read/write head rather than by actual mechanical movement; consequently, it is much faster. However, the cost of the additional read/write heads is high, so fixed-head disks are not commonly used.

**Interfacing Disk Drives to Computer Systems.** 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 for disk drives on PCs and workstations was called **SCSI** (small computer system interface). Today to connect HDDs, CDs, and DVDs to a computer, the interface of choice is **SATA**. **SATA** stands for serial ATA, wherein ATA represents attachment; so SATA becomes serial AT attachment. It has its origin in PC/AT attachment, which referred to the direct attachment to the 16-bit bus introduced by IBM. The AT referred to advanced technology but is not used in the expansion of SATA due to trademark issues. Another popular interface used today is called **SAS** (serial attached SCSI). SATA was introduced in 2002 and allows the disk controller to be in the disk drive; only a simple circuit is required on the motherboard. SATA transfer speeds underwent an evolution from 2002 to 2008, going from 1.5 Gbps (gigabits per second) to 6 Gbps. SATA is now called **NL-SAS** for nearline SAS. The largest 3.5-inch SATA and SAS drives are 8TB, whereas 2.5-inch SAS drives are smaller and go up to 1.2TB. The 3.5-inch drives use 7,200 or 10,000 rpm speed whereas 2.5-inch drives use up to 15,000 rpm. In terms of IOPs (input/output operations) per second as a price to performance index, SAS is considered superior to SATA.

The controller accepts high-level I/O commands and takes appropriate action to position the arm and causes the read/write action to take place. To transfer a disk block, given its address, the disk controller must first mechanically position the

read/write head on the correct track. The time required to do this is called the **seek time**. Typical seek times are 5 to 10 msec on desktops and 3 to 8 msec on servers. Following that, there is another delay—called the **rotational delay** or **latency**—while the beginning of the desired block rotates into position under the read/write head. It depends on the rpm of the disk. For example, at 15,000 rpm, the time per rotation is 4 msec and the average rotational delay is the time per half revolution, or 2 msec. At 10,000 rpm the average rotational delay increases to 3 msec. Finally, some additional time is needed to transfer the data; this is called the **block transfer time**. Hence, the total time needed to locate and transfer an arbitrary block, given its address, is the sum of the seek time, rotational delay, and block transfer time. The seek time and rotational delay are usually much larger than the block transfer time. To make the transfer of multiple blocks more efficient, it is common to transfer several consecutive blocks on the same track or cylinder. This eliminates the seek time and rotational delay for all but the first block and can result in a substantial saving of time when numerous contiguous blocks are transferred. Usually, the disk manufacturer provides a **bulk transfer rate** for calculating the time required to transfer consecutive blocks. Appendix B contains a discussion of these and other disk parameters.

The time needed to locate and transfer a disk block is in the order of milliseconds, usually ranging from 9 to 60 msec. For contiguous blocks, locating the first block takes from 9 to 60 msec, but transferring subsequent blocks may take only 0.4 to 2 msec each. Many search techniques take advantage of consecutive retrieval of blocks when searching for data on a disk. In any case, a transfer time in the order of milliseconds is considered 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 17 attempt to *minimize the number of block transfers* needed to locate and transfer the required data from disk to main memory. Placing “related information” on contiguous blocks is the basic goal of any storage organization on disk.

## 16.2.2 Making Data Access More Efficient on Disk

In this subsection, we list some of the commonly used techniques to make accessing data more efficient on HDDs.

1. **Buffering of data:** In order to deal with the incompatibility of speeds between a CPU and the electromechanical device such as an HDD, which is inherently slower, buffering of data is done in memory so that new data can be held in a buffer while old data is processed by an application. We discuss the double buffering strategy followed by general issues of buffer management and buffer replacement strategies in Section 16.3.
2. **Proper organization of data on disk:** Given the structure and organization of data on disk, it is advantageous to keep related data on contiguous blocks; when multiple cylinders are needed by a relation, contiguous cylinders should be used. Doing so avoids unnecessary movement of the read/write arm and related seek times.

3. **Reading data ahead of request:** To minimize seek times, whenever a block is read into the buffer, blocks from the rest of the track can also be read even though they may not have been requested yet. This works well for applications that are likely to need consecutive blocks; for random block reads this strategy is counterproductive.
4. **Proper scheduling of I/O requests:** If it is necessary to read several blocks from disk, total access time can be minimized by scheduling them so that the arm moves only in one direction and picks up the blocks along its movement. One popular algorithm is called the elevator algorithm; this algorithm mimics the behavior of an elevator that schedules requests on multiple floors in a proper sequence. In this way, the arm can service requests along its outward and inward movements without much disruption.
5. **Use of log disks to temporarily hold writes:** A single disk may be assigned to just one function called logging of writes. All blocks to be written can go to that disk sequentially, thus eliminating any seek time. This works much faster than doing the writes to a file at random locations, which requires a seek for each write. The log disk can order these writes in (cylinder, track) ordering to minimize arm movement when writing. Actually, the log disk can only be an area (extent) of a disk. Having the data file and the log file on the same disk is a cheaper solution but compromises performance. Although the idea of a log disk can improve write performance, it is not feasible for most real-life application data.
6. **Use of SSDs or flash memory for recovery purposes:** In applications where updates occur with high frequency, updates can be lost from main memory if the system crashes. A preventive measure would be to increase the speed of updates/writes to disk. One possible approach involves writing the updates to a nonvolatile SSD buffer, which may be a flash memory or battery-operated DRAM, both of which operate at much faster speeds (see Table 16.1). The disk controller then updates the data file during its idle time and also when the buffer becomes full. During recovery from a crash, unwritten SSD buffers must be written to the data file on HDD. For further discussion of recovery and logs, consult Chapter 22.

### 16.2.3 SolidState Device (SSD) Storage

This type of storage is sometimes known as flash storage because it is based on the flash memory technology, which we discussed in Section 16.1.1.

The recent trend is to use flash memories as an intermediate layer between main memory and secondary rotating storage in the form of magnetic disks (HDDs). Since they resemble disks in terms of the ability to store data in secondary storage without the need for continuous power supply, they are called **solid-state disks** or **solid-state drives (SSDs)**. We will discuss SSDs in general terms first and then comment on their use at the enterprise level, where they are sometimes referred to as **enterprise flash drives (EFDs)**, a term first introduced by EMC Corporation.

The main component of an SSD is a controller and a set of interconnected flash memory cards. Use of NAND flash memory is most common. Using form factors compatible with 3.5 inch or 2.5 inch HDDs makes SSDs pluggable into slots already available for mounting HDDs on laptops and servers. For ultrabooks, tablets, and the like, card-based form factors such as mSATA and M.2 are being standardized. Interfaces like **SATA express** have been created to keep up with advancements in SSDs. Because there are no moving parts, the unit is more rugged, runs silently, is faster in terms of access time and provides higher transfer rates than HDD. As opposed to HDDs, where related data from the same relation must be placed on contiguous blocks, preferably on contiguous cylinders, there is no restriction on placement of data on an SSD since any address is directly addressable. As a result, the data is less likely to be fragmented; hence no reorganization is needed. Typically, when a write to disk occurs on an HDD, the same block is overwritten with new data. In SSDs, the data is written to different NAND cells to attain **wear-leveling**, which prolongs the life of the SSD. The main issue preventing a wide-scale adoption of SSDs today is their prohibitive cost (see Table 16.1), which tends to be about 70 to 80 cents per GB as opposed to about 15 to 20 cents per GB for HDDs.

In addition to flash memory, DRAM-based SSDs are also available. They are costlier than flash memory, but they offer faster access times of around 10  $\mu$ s (microseconds) as opposed to 100  $\mu$ s for flash. Their main drawback is that they need an internal battery or an adapter to supply power.

As an example of an enterprise level SSD, we can consider CISCO's UCS (Unified Computing System<sup>®</sup>) Invicta series SSDs. They have made it possible to deploy SSDs at the data center level to unify workloads of all types, including databases and virtual desktop infrastructure (VDI), and to enable a cost-effective, energy-efficient, and space-saving solution. CISCO's claim is that Invicta SSDs offer a better price-to-performance ratio to applications in a multitenant, multinetworked architecture because of the advantages of SSDs stated above. CISCO states that typically four times as many HDD drives may be needed to match an SSD-based RAID in performance.<sup>6</sup> The SSD configuration can have a capacity from 6 to 144 TB, with up to 1.2 million I/O operations/second, and a bandwidth of up to 7.2 GB/sec with an average latency of 200  $\mu$ s.<sup>7</sup> Modern data centers are undergoing rapid transformation and must provide real-time response using cloud-based architectures. In this environment, SSDs are likely to play a major role.

### 16.2.4 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

---

<sup>6</sup>Based on the CISCO White Paper (CISCO, 2014)

<sup>7</sup>Data sheet for CISCO UCS Invicta Scaling System.

$n - 1$  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 1,600 to 6,250 bytes per inch, a typical interblock gap<sup>8</sup> of 0.6 inch corresponds to 960 to 3,750 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 recordkeeping can be **archived** on tape. Originally, half-inch reel tape drives were used for data storage employing the so-called nine-track tapes. Later, smaller 8-mm magnetic tapes (similar to those used in camcorders) that can store up to 50 GB, as well as 4-mm helical scan data cartridges and writable CDs and DVDs, became popular media for backing up data files from PCs and workstations. They are also used for storing images and system libraries.

Backing up enterprise databases so that no transaction information is lost is a major undertaking. Tape libraries were in vogue and featured slots for several hundred cartridges; these tape libraries used digital and superdigital linear tapes (DLTs and SDLTs), both of which have capacities in the hundreds of gigabytes and record data on linear tracks. These tape libraries are no longer in further development. The LTO (Linear Tape Open) consortium set up by IBM, HP, and Seagate released the latest LTO-6 standard in 2012 for tapes. It uses ½-inch-wide magnetic tapes like those used in earlier tape drives but in a somewhat smaller, single-reel enclosed cartridge. Current generation of libraries use LTO-6 drives, at 2.5-TB cartridge with 160 MB/s transfer rate. Average seek time is about 80 seconds. The T10000D drive of Oracle/StorageTek handles 8.5 TB on a single cartridge with transfer rate upto 252 MB/s.

---

<sup>8</sup>Called *interrecord gaps* in tape terminology.



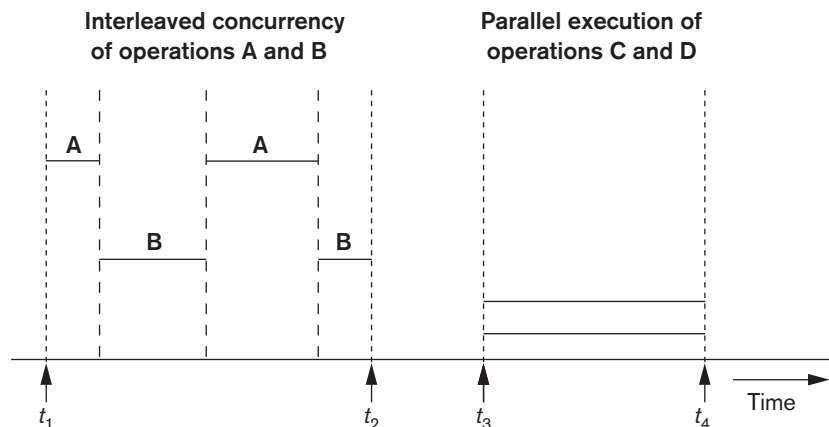
Robotic arms write on multiple cartridges in parallel using multiple tape drives and automatic labeling software to identify the backup cartridges. An example of a giant library is the SL8500 model of Sun Storage Technology. The SL8500 scales from 1,450 to just over 10,000 slots and from 1 to 64 tape drives within each library. It accepts both DLT/SDLT and LTO tapes. Up to 10 SL8500s can be connected within a single library complex for over 100,000 slots and up to 640 drives. With 100,000 slots, the SL8500 can store 2.1 exabytes (exabyte = 1,000 petabytes, or million TB =  $10^{18}$  bytes). We defer the discussion of disk storage technology called RAID, and of storage area networks, network-attached storage, and iSCSI storage systems, to the end of the chapter.

## 16.3 Buffering of Blocks

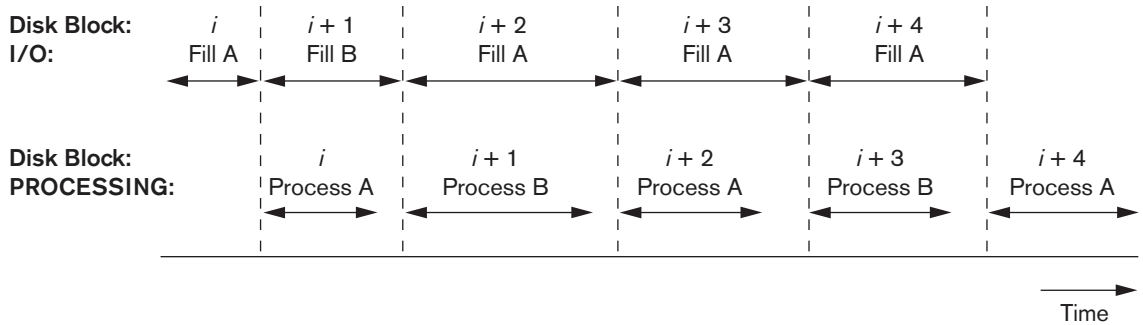
When several blocks need to be transferred from disk to main memory and all the block addresses are known, several buffers can be reserved in main memory to speed up the transfer. While one buffer is being read or written, the CPU can process data in the other buffer because an independent disk I/O processor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

Figure 16.3 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 16.4 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



**Figure 16.3**  
Interleaved concurrency  
versus parallel execution.

**Figure 16.4**

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

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

### 16.3.1 Buffer Management

**Buffer management and Replacement Strategies.** For most large database files containing millions of pages, it is not possible to bring all of the data into main memory at the same time. We alluded to double buffering as a technique whereby we can gain efficiency in terms of performing the I/O operation between the disk and main memory into one buffer area concurrently with processing the data from another buffer. The actual management of buffers and decisions about what buffers to use to place a newly read page in the buffer is a more complex process. We use the term **buffer** to refer to a part of main memory that is available to receive blocks or pages of data from disk.<sup>9</sup> **Buffer manager** is a software component of a DBMS that responds to requests for data and decides what buffer to use and what pages to replace in the buffer to accommodate the newly requested blocks. The buffer manager views the available main memory storage as a **buffer pool**, which has a collection of pages. The size of the shared buffer pool is typically a parameter for the DBMS controlled by DBAs. In this section, we briefly discuss the workings of the buffer manager and discuss a few replacement strategies.

<sup>9</sup>We use the terms page and block interchangeably in the current context.

There are two kinds of buffer managers; the first kind controls the main memory directly, as in most RDBMSs. The second kind allocates buffers in virtual memory, which allows the control to transfer to the operating system (OS). The OS in turn controls which buffers are actually in main memory and which ones are on disk under the control of OS. This second kind of buffer manager is common in main memory database systems and some object-oriented DBMSs. The overall goal of the buffer manager is twofold: (1) to maximize the probability that the requested page is found in main memory, and (2) in case of reading a new disk block from disk, to find a page to replace that will cause the least harm in the sense that it will not be required shortly again.

To enable its operation, the buffer manager keeps two types of information on hand about each page in the buffer pool:

1. A **pin-count**: the number of times that page has been requested, or the number of current users of that page. If this count falls to zero, the page is considered **unpinned**. Initially the pin-count for every page is set to zero. Incrementing the pin-count is called **pinning**. In general, a pinned block should not be allowed to be written to disk.
2. A **dirty bit**, which is initially set to zero for all pages but is set to 1 whenever that page is updated by any application program.

In terms of storage management, the buffer manager has the following responsibility: It must make sure that the number of buffers fits in main memory. If the requested amount of data exceeds available buffer space, the buffer manager must select what buffers must be emptied, as governed by the buffer replacement policy in force. If the buffer manager allocates space in virtual memory and all buffers in use exceed the actual main memory, then the common operating system problem of “thrashing” happens and pages get moved back and forth into the swap space on disk without performing useful work.

When a certain page is requested, the buffer manager takes following actions: it checks if the requested page is already in a buffer in the buffer pool; if so, it increments its pin-count and releases the page. If the page is not in the buffer pool, the buffer manager does the following:

- a. It chooses a page for replacement, using the replacement policy, and increments its pin-count.
- b. If the dirty bit of the replacement page is on, the buffer manager writes that page to disk by replacing its old copy on disk. If the dirty bit is not on, this page is not modified and the buffer manager is not required to write it back to disk.
- c. It reads the requested page into the space just freed up.
- d. The main memory address of the new page is passed to the requesting application.

If there is no unpinned page available in the buffer pool and the requested page is not available in the buffer pool, the buffer manager may have to wait until a page gets released. A transaction requesting this page may go into a wait state or may even be aborted.

### 16.3.2 Buffer Replacement Strategies:

The following are some popular replacement strategies that are similar to those used elsewhere, such as in operating systems:

1. **Least recently used (LRU):** The strategy here is to throw out that page that has not been used (read or written) for the longest time. This requires the buffer manager to maintain a table where it records the time every time a page in a buffer is accessed. Whereas this constitutes an overhead, the strategy works well because for a buffer that is not used for a long time, its chance of being accessed again is small.
2. **Clock policy:** This is a round-robin variant of the LRU policy. Imagine the buffers are arranged like a circle similar to a clock. Each buffer has a flag with a 0 or 1 value. Buffers with a 0 are vulnerable and may be used for replacement and their contents read back to disk. Buffers with a 1 are not vulnerable. When a block is read into a buffer, the flag is set to 1. When the buffer is accessed, the flag is set to 1 also. The clock hand is positioned on a “current buffer.” When the buffer manager needs a buffer for a new block, it rotates the hand until it finds a buffer with a 0 and uses that to read and place the new block. (If the dirty bit is on for the page being replaced, that page will be written to disk, thus overwriting the old page at its address on disk.) If the clock hand passes buffers with 1s, it sets them to a zero. Thus, a block is replaced from its buffer only if it is not accessed until the hand completes a rotation and returns to it and finds the block with the 0 that it set the last time.
3. **First-in-first-out (FIFO):** Under this policy, when a buffer is required, the one that has been occupied the longest by a page is used for replacement. Under this policy, the manager notes the time each page gets loaded into a buffer; but it does not have to keep track of the time pages are accessed. Although FIFO needs less maintenance than LRU, it can work counter to desirable behavior. A block that remains in the buffer for a long time because it is needed continuously, such as a root block of an index, may be thrown out but may be immediately required to be brought back.

LRU and clock policies are not the best policies for database applications if they require sequential scans of data and the file cannot fit into the buffer at one time. There are also situations when certain pages in buffers cannot be thrown out and written out to disk because certain other pinned pages point to those pages. Also, policies like FIFO can be modified to make sure that pinned blocks, such as root block of an index, are allowed to remain in the buffer. Modification of the clock policy also exists where important buffers can be set to higher values than 1 and therefore will not be subjected to replacement for several rotations of the hand. There are also situations when the DBMS has the ability to write certain blocks to disk even when the space occupied by those blocks is not needed. This is called **force-writing** and occurs typically when log records have to be written to disk ahead of the modified pages in a transaction for recovery purposes. (See Chapter 22.) There are some other replacement strategies such as **MRU (most recently used)**

that work well for certain types of database transactions, such as when a block that is used most recently is not needed until all the remaining blocks in the relation are processed.

## 16.4 Placing File Records on Disk

Data in a database is regarded as a set of records organized into a set of files. In this section, we define the concepts of records, record types, and files. Then we discuss techniques for placing file records on disk. Note that henceforth in this chapter we will be referring to the random access persistent secondary storage as “disk drive” or “disk.” The disk may be in different forms; for example, magnetic disks with rotational memory or solid-state disks with electronic access and no mechanical delays.

### 16.4.1 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 some database applications, the need may arise for storing data items that consist of large unstructured objects, which represent images, digitized video or audio streams, or free text. These are referred to as **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. For storing free text, some DBMSs (e.g., Oracle, DB2, etc.) provide a data type called CLOB (character large object); some DBMSs call this data type text.

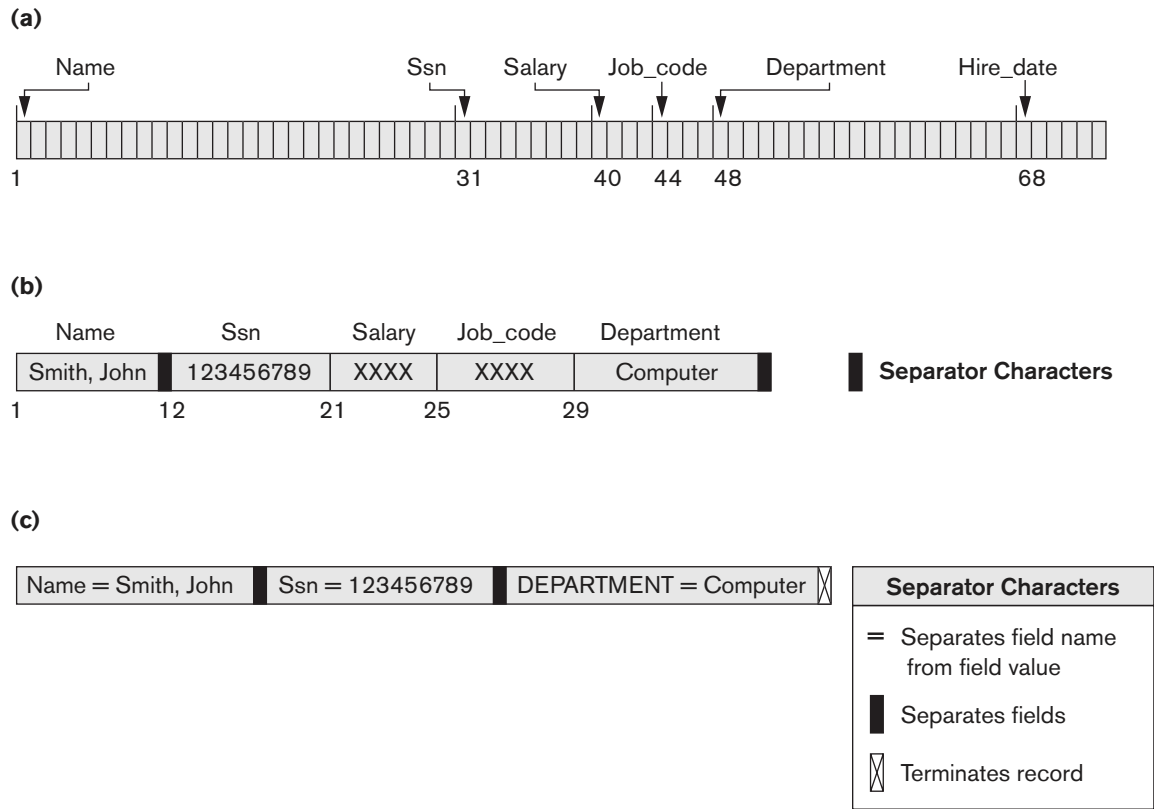
### 16.4.2 Files, Fixed-Length Records, and Variable-Length Records

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 16.5(a) have a record size of 71 bytes. Every record has the same fields, and field lengths are fixed, so the system can identify the starting byte position of each field relative to the starting position of the record. This facilitates locating field values by programs that access such files. Notice that it is possible to represent a file that logically should have variable-length records as a fixed-length records file. For example, in the case of optional fields, we could have *every field* included in *every file record* but store a special NULL value if no value exists for that field. For a repeating field, we could allocate as many spaces in each record as the *maximum possible number of occurrences* of the field. In either case, space is wasted when certain records do not have values for all the physical spaces provided in each record. Now we consider other options for formatting records of a file of variable-length records.

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 16.5(b), or we can store the length in bytes of the field in the record, preceding the field value.



**Figure 16.5** 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.

A file of records with *optional fields* can be formatted in different ways. If the total number of fields for the record type is large, but the number of fields that actually appear in a typical record is small, we can include in each record a sequence of <field-name, field-value> pairs rather than just the field values. Three types of separator characters are used in Figure 16.5(c), although we could use the same separator character for the first two purposes—separating the field name from the field value and separating one field from the next field. A more practical option is to assign a short **field type** code—say, an integer number—to each field and include in each record a sequence of <field-type, field-value> pairs rather than <field-name, field-value> pairs.

A *repeating field* needs one separator character to separate the repeating values of the field and another separator character to indicate termination of the field. Finally, for a file that includes *records of different types*, each record is preceded by a **record**



**type** indicator. Understandably, programs that process files of variable-length records—which are usually part of the file system and hence hidden from the typical programmers—need to be more complex than those for fixed-length records, where the starting position and size of each field are known and fixed.<sup>10</sup>

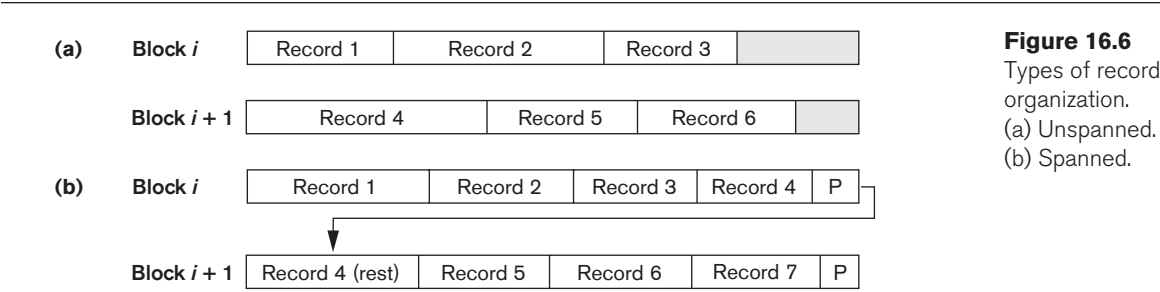
### 16.4.3 Record Blocking and Spanned versus Unspanned Records

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 16.6 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*



<sup>10</sup>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.

#### 16.4.4 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.

#### 16.4.5 File Headers

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

To search for a record on disk, one or more blocks are copied into main memory buffers. Programs then search for the desired record or records within the buffers, using the information in the file header. If the address of the block that contains the desired record is not known, the search programs must do a **linear search** through 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 avoid linear search or full scan of the file and to locate the block that contains a desired record with a minimal number of block transfers.

### 16.5 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. In the following list, we present a set of representative operations. Typically, high-level programs, such as DBMS software programs, access records by using these commands, so we sometimes refer to **program variables** in the following descriptions:

- **Open.** Prepares the file for reading or writing. Allocates appropriate buffers (typically at least two) to hold file blocks from disk, and retrieves the file header. Sets the file pointer to the beginning of the file.
- **Reset.** Sets the file pointer of an open file to the beginning of the file.
- **Find (or Locate).** Searches for the first record that satisfies a search condition. Transfers the block containing that record into a main memory buffer (if it is not already there). The file pointer points to the record in the buffer and it becomes the *current record*. Sometimes, different verbs are used to indicate whether the located record is to be retrieved or updated.
- **Read (or Get).** Copies the current record from the buffer to a program variable in the user program. This command may also advance the current record pointer to the next record in the file, which may necessitate reading the next file block from disk.
- **FindNext.** Searches for the next record in the file that satisfies the search condition. Transfers the block containing that record into a main memory buffer (if it is not already there). The record is located in the buffer and becomes the current record. Various forms of FindNext (for example, FindNext record within a current parent record, FindNext record of a given type, or FindNext record where a complex condition is met) are available in legacy DBMSs based on the hierarchical and network models.

- **Delete.** Deletes the current record and (eventually) updates the file on disk to reflect the deletion.
- **Modify.** Modifies some field values for the current record and (eventually) updates the file on disk to reflect the modification.
- **Insert.** Inserts a new record in the file by locating the block where the record is to be inserted, transferring that block into a main memory buffer (if it is not already there), writing the record into the buffer, and (eventually) writing the buffer to disk to reflect the insertion.
- **Close.** Completes the file access by releasing the buffers and performing any other needed cleanup operations.

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

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

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

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

At this point, it is worthwhile to note the difference between the terms *file organization* and *access method*. A **file organization** refers to the organization of the data of a file into records, blocks, and access structures; this includes the way records and blocks are placed on the storage medium and interlinked. An **access method**, on the other hand, provides a group of operations—such as those listed earlier—that can be applied to a file. In general, it is possible to apply several access methods to a file organized using a certain 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 17).

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. If a file is not updatable by the end user, it is regarded as a read-only file.

Most data warehouses (see Chapter 29) predominantly contain read-only files. 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 16.5(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 17). Suppose that a second application uses the file to generate employees' paychecks and requires that paychecks are grouped by department. For this application, it is best to order employee records by department and then by name within each department. The clustering of records into blocks and the organization of blocks on cylinders would now be different than before. However, this arrangement conflicts with ordering the records by Ssn values. If both applications are important, the designer should choose an organization that allows both operations to be done efficiently. Unfortunately, in many cases a single organization does not allow all needed operations on a file to be implemented efficiently. Since a file can be stored only once using one particular organization, the DBAs are often faced with making a difficult design choice about the file organization. They make it based on the expected importance and mix of retrieval and update operations.

In the following sections and in Chapter 17, 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.

## 16.6 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**.<sup>11</sup> This organization is often used with additional access paths, such as the secondary indexes discussed in Chapter 17. 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

---

<sup>11</sup>Sometimes this organization is called a **sequential file**.

blocks before it finds the record. For a file of  $b$  blocks, this requires searching  $(b/2)$  blocks, on average. If no records or several records satisfy the search condition, the program must read and search all  $b$  blocks in the file.

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

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

To read all records in order of the values of some field, we create a sorted copy of the file. Sorting is an expensive operation for a large disk file, and special techniques for **external sorting** are used (see Chapter 18).

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  $0, 1, 2, \dots, r - 1$  and the records in each block are numbered  $0, 1, \dots, bfr - 1$ , where  $bfr$  is the blocking factor, then the  $i$ th record of the file is located in block  $\lfloor (i/bfr) \rfloor$  and is the  $(i \bmod bfr)$ th record in that block. Such a file is often called a **relative** or **direct file** because records can easily be accessed directly by their relative 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 17.

## 16.7 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**.<sup>12</sup> 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 16.7

<sup>12</sup>The term *sequential file* has also been used to refer to unordered files, although it is more appropriate for ordered 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 16.7**

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



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. The search condition may be of the type  $\langle \text{key} = \text{value} \rangle$ , or a range condition such as  $\langle \text{value1} < \text{key} < \text{value2} \rangle$ . Second, finding the next record from the current one in order of the ordering key usually requires no additional block accesses because the next record is in the same block as the current one (unless the current record is the last one in the block). Third, using a search condition based on the value of an ordering key field results in faster access when the binary search technique is used, which constitutes an improvement over linear searches, although it is not often used for disk files. Ordered files are blocked and stored on contiguous cylinders to minimize the seek time.

A **binary search** for disk files can be done on the blocks rather than on the records. Suppose that the file has  $b$  blocks numbered 1, 2, ...,  $b$ ; the records are ordered by ascending value of their ordering key field; and we are searching for a record whose ordering key field value is  $K$ . Assuming that disk addresses of the file blocks are available in the file header, the binary search can be described by Algorithm 16.1. 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 16.1.** 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  $i \leftarrow (l + u) \text{ div } 2$ ;
    read block  $i$  of the file into the buffer;
    if  $K < (\text{ordering key field value of the first record in block } i)$ 
        then  $u \leftarrow i - 1$ 
    else if  $K > (\text{ordering key field value of the last record in block } i)$ 
        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 efficient, since the physical ordering of records means that all records satisfying the condition are contiguous in the file. For example, referring to Figure 16.7, if the search criterion is (Name  $>$  'G')—where  $>$  means *alphabetically before*—the records satisfying the search criterion are those from the beginning of the file up to the first record that has a Name value starting with the letter 'G'.

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 non-ordering 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. One option is to keep the highest value of the key in each block in a separate field after taking into account the keys that have overflowed from that block. Otherwise, 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 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 16.3 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**.

**Table 16.3** 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	$\log_2 b$

This further improves the random access time on the ordering key field. (We discuss indexes in Chapter 17.) If the ordering attribute is not a key, the file is called a **clustered file**.

## 16.8 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**.<sup>13</sup> The search condition must be an equality condition on a single field, called the **hash field**. In most cases, the hash field is also a key field of the file, in which case it is called the **hash key**. The idea behind hashing is to provide a function  $h$ , called a **hash function** or **randomizing function**, which is applied to the hash field value of a record and yields the *address* of the disk block in which the record is stored. A search for the record within the block can be carried out in a main memory buffer. For most records, we need only a single-block access to retrieve that record.

Hashing is also used as an internal search structure within a program whenever a group of records is accessed exclusively by using the value of one field. We describe the use of hashing for internal files in Section 16.8.1; then we show how it is modified to store external files on disk in Section 16.8.2. In Section 16.8.3 we discuss techniques for extending hashing to dynamically growing files.

### 16.8.1 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 16.8(a); then we have  $M$  **slots** whose addresses correspond to the array indexes. We choose a hash function that transforms the hash field value into an integer between 0 and  $M - 1$ . One common hash function is the  $h(K) = K \bmod M$  function, which returns the remainder of an integer hash field value  $K$  after division by  $M$ ; this value is then used for the record address.

<sup>13</sup>A hash file has also been called a *direct file*.



**Algorithm 16.2.** 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  $temp \leftarrow temp * code(K[i]) \bmod M$ ;
     $hash\_address \leftarrow temp \bmod M$ ;

(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;
```

Other hashing functions can be used. One technique, called **folding**, involves applying an arithmetic function such as *addition* or a logical function such as *exclusive or* to different portions of the hash field value to calculate the hash address (for example, with an address space from 0 to 999 to store 1,000 keys, a 6-digit key 235469 may be folded and stored at the address:  $(235+964) \bmod 1000 = 199$ ). Another technique involves picking some digits of the hash field value—for instance, the third, fifth, and eighth digits—to form the hash address (for example, storing 1,000 employees with Social Security numbers of 10 digits into a hash file with 1,000 positions would give the Social Security number 301-67-8923 a hash value of 172 by this hash function).<sup>14</sup> 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 16.2(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.

<sup>14</sup> A detailed discussion of hashing functions is outside the scope of our presentation.

A linked list of overflow records for each hash address is thus maintained, as shown in Figure 16.8(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. Note that the series of hash functions are used in the same order for retrieval.

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.

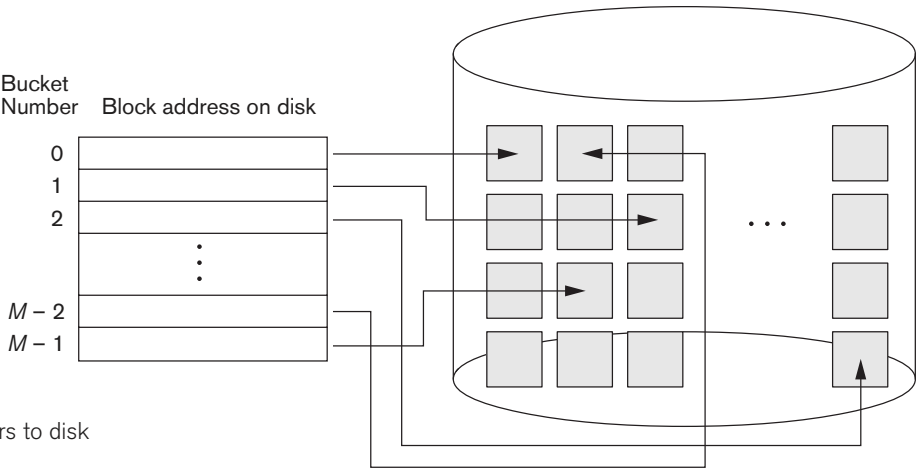
The goal of a good hashing function is twofold: first, to distribute the records uniformly over the address space so as to minimize collisions, thus making it possible to locate a record with a given key in a single access. The second, somewhat conflicting, goal is to achieve the above yet occupy the buckets fully, thus not leaving many unused locations. Simulation and analysis studies have shown that it is usually best to keep a hash file between 70 and 90% full so that the number of collisions remains low and we do not waste too much space. Hence, if we expect to have  $r$  records to store in the table, we should choose  $M$  locations for the address space such that  $(r/M)$  is between 0.7 and 0.9. It may also be useful to choose a prime number for  $M$ , since it has been demonstrated that this distributes the hash addresses better over the address space when the mod hashing function is used modulo a prime number. Other hash functions may require  $M$  to be a power of 2.

## 16.8.2 External Hashing for Disk Files

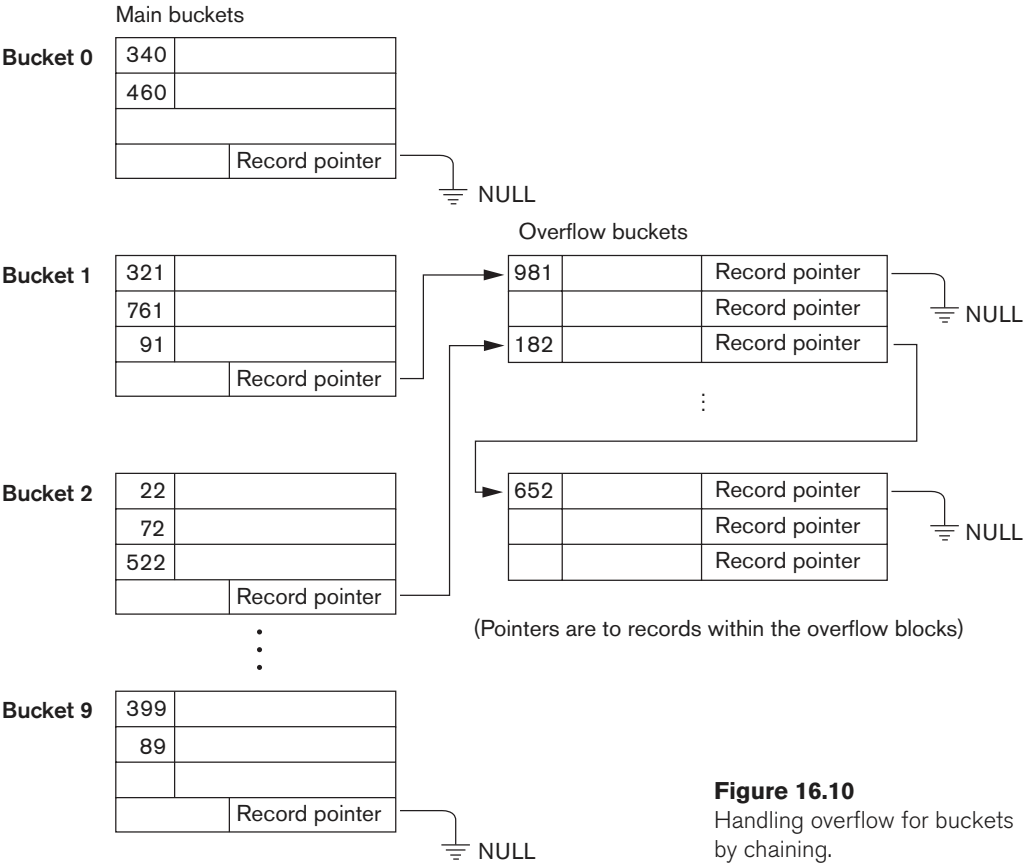
Hashing for disk files is called **external hashing**. To suit the characteristics of disk storage, the target address space is made of **buckets**, each of which holds multiple records. A bucket is either one disk block or a cluster of contiguous disk blocks. The hashing function maps a key into a relative bucket number rather than assigning an absolute block address to the bucket. A table maintained in the file header converts the bucket number into the corresponding disk block address, as illustrated in Figure 16.9.

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 16.10. 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



**Figure 16.9**  
Matching bucket numbers to disk block addresses.



**Figure 16.10**  
Handling overflow for buckets by chaining.



records in order of hash field values, some functions—called **order preserving**—do. A simple example of an order-preserving hash function is to take the leftmost three digits of an invoice number field that yields a bucket address as the hash address and keep the records sorted by invoice number within each bucket. Another example is to use an integer hash key directly as an index to a relative file, if the hash key values fill up a particular interval; for example, if employee numbers in a company are assigned as 1, 2, 3, ... up to the total number of employees, we can use the identity hash function (i.e., Relative Address = Key) that maintains order. Unfortunately, this only works if sequence keys are generated in order by some application.

The hashing scheme described so far is called **static hashing** because a fixed number of buckets  $M$  is allocated. The function does key-to-address mapping, whereby we are fixing the address space. 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 16.8.3).

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

### 16.8.3 Hashing Techniques That Allow Dynamic File Expansion

A major drawback of the *static* hashing scheme just discussed is that the hash address space is fixed. Hence, it is difficult to expand or shrink the file dynamically. The schemes described in this section attempt to remedy this situation. The first

scheme—extendible hashing—stores an access structure in addition to the file, and hence is somewhat similar to indexing (see Chapter 17). The main difference is that the access structure is based on the values that result after application of the hash function to the search field. In indexing, the access structure is based on the values of the search field itself. The second technique, called linear hashing, does not require additional access structures. Another scheme, called **dynamic hashing**, uses an access structure based on binary tree data structures.

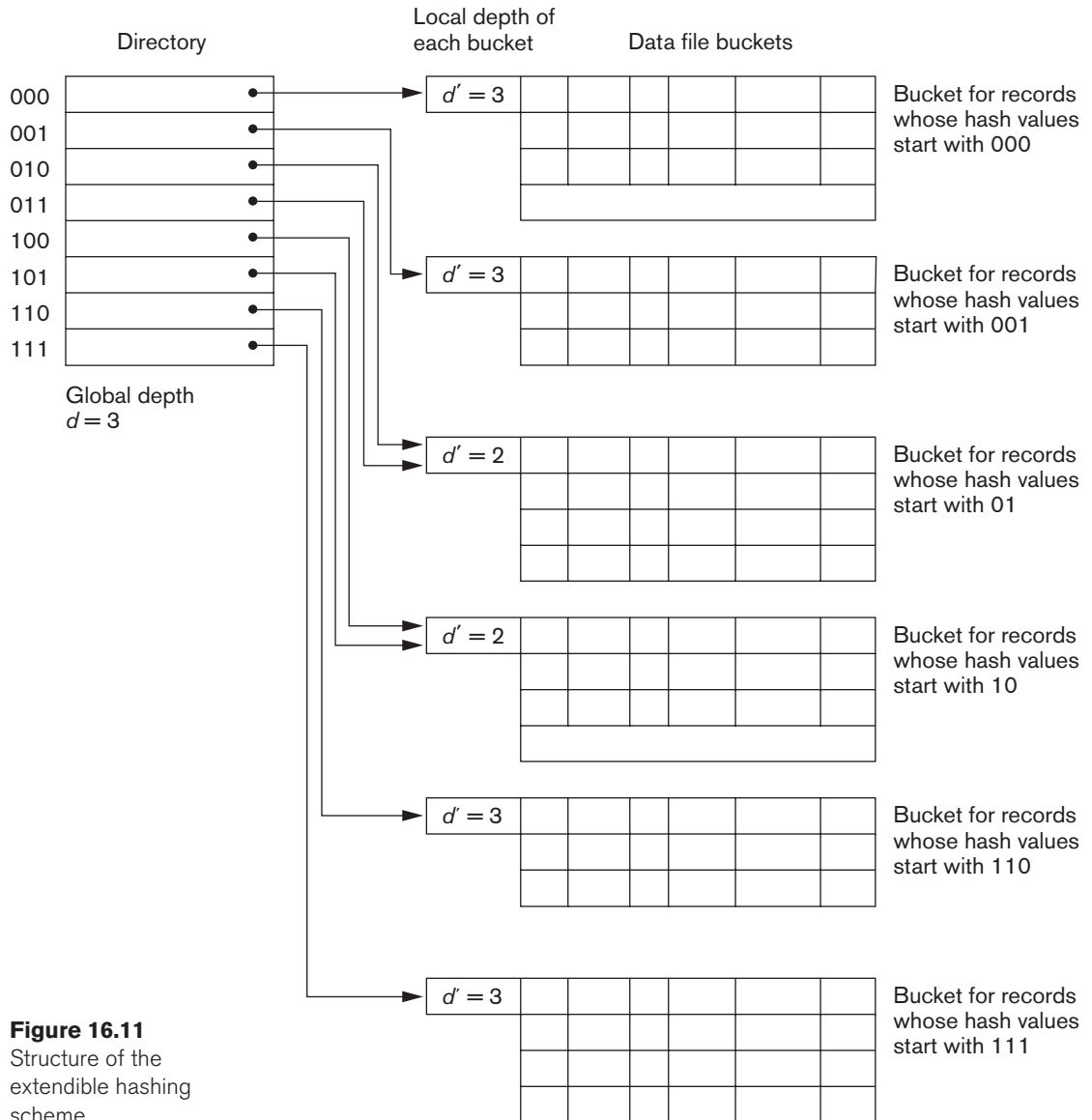
These hashing schemes take advantage of the fact that the result of applying a hashing function is a nonnegative integer and hence can be represented as a binary number. The access structure is built on the **binary representation** of the hashing function result, which is a string of **bits**. We call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

**Extendible Hashing.** In extendible hashing, proposed by Fagin (1979), 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 16.11 shows a directory with global depth  $d = 3$ .

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

To illustrate bucket splitting, suppose that a new inserted record causes overflow in the bucket whose hash values start with 01—the third bucket in Figure 16.11. The records will be distributed between two buckets: the first contains all records whose hash values start with 010, and the second all those whose hash values start with 011. Now the two directory locations for 010 and 011 point to the two new distinct buckets. Before the split, they pointed to the same bucket. The local depth  $d'$  of the two new buckets is 3, which is one more than the local depth of the old bucket.

If a bucket that overflows and is split used to have a local depth  $d'$  equal to the global depth  $d$  of the directory, then the size of the directory must now be doubled so that we can use an extra bit to distinguish the two new buckets. For example, if the bucket for records whose hash values start with 111 in Figure 16.11 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 effectively increases the average number of accesses per key. Additionally, no space is allocated in extendible hashing for future growth, but additional buckets can be allocated

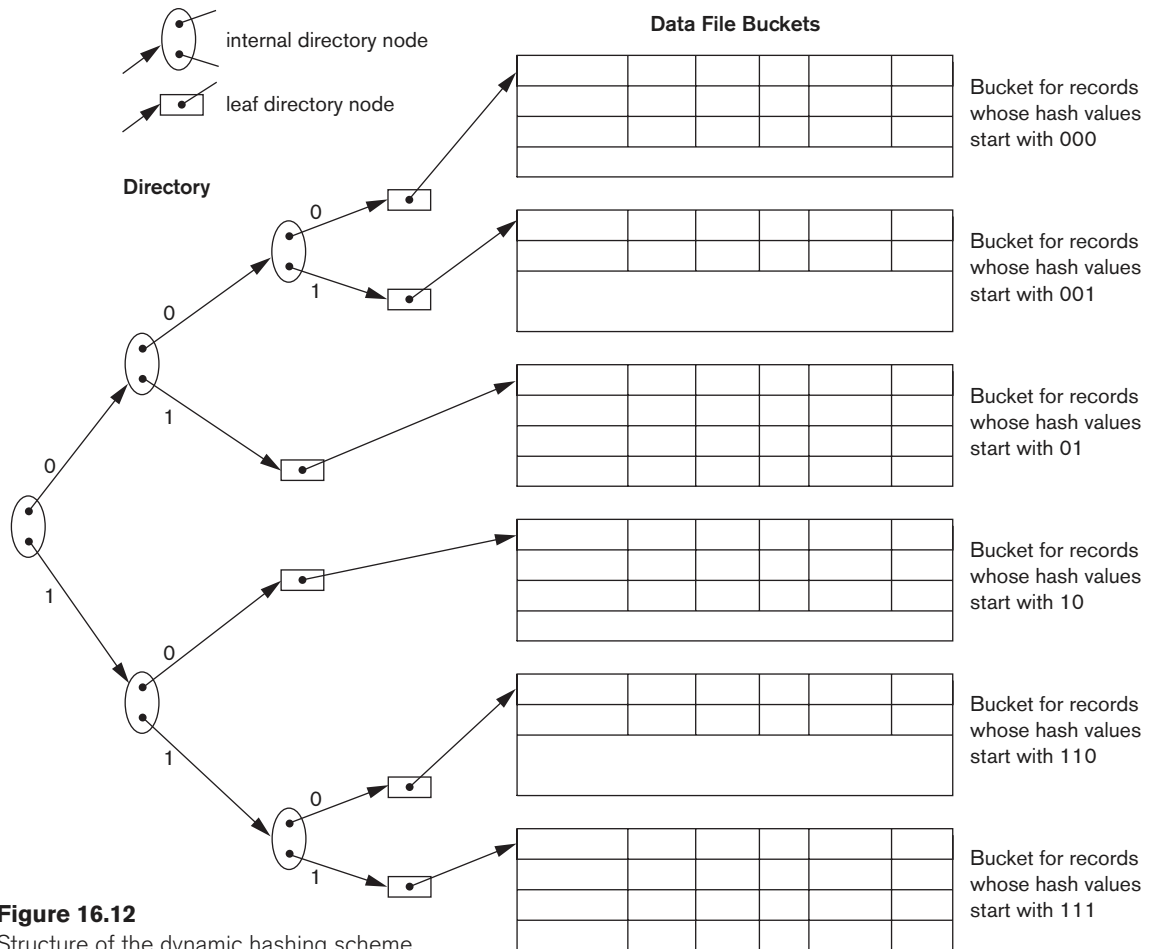
dynamically as needed. The space overhead for the directory table is negligible. The maximum directory size is  $2^k$ , where  $k$  is the number of bits in the hash value. Another advantage is that splitting causes minor reorganization in most cases, since only the records in one bucket are redistributed to the two new buckets. The only time reorganization is more expensive is when the directory has to be doubled (or halved). A disadvantage is that the directory must be searched before accessing the buckets themselves, resulting in two block accesses instead of one in static hashing. This performance penalty is considered minor and thus the scheme is considered quite desirable for dynamic files.

**Dynamic Hashing.** A precursor to extendible hashing was dynamic hashing proposed by Larson (1978), in which the addresses of the buckets were either the  $n$  high-order bits or  $n - 1$  high-order bits, depending on the total number of keys belonging to the respective bucket. The eventual storage of records in buckets for dynamic hashing is somewhat similar to extendible hashing. The major difference is in the organization of the directory. Whereas extendible hashing uses the notion of global depth (high-order  $d$  bits) for the flat directory and then combines adjacent collapsible buckets into a bucket of local depth  $d - 1$ , dynamic hashing maintains a tree-structured directory with two types of nodes:

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

An example of the dynamic hashing appears in Figure 16.12. Four buckets are shown (“000”, “001”, “110”, and “111”) with high-order 3-bit addresses (corresponding to the global depth of 3), and two buckets (“01” and “10”) are shown with high-order 2-bit addresses (corresponding to the local depth of 2). The latter two are the result of collapsing the “010” and “011” into “01” and collapsing “100” and “101” into “10”. Note that the directory nodes are used implicitly to determine the “global” and “local” depths of buckets in dynamic hashing. The search for a record given the hashed address involves traversing the directory tree, which leads to the bucket holding that record. It is left to the reader to develop algorithms for insertion, deletion, and searching of records for the dynamic hashing scheme.

**Linear Hashing.** The idea behind linear hashing, proposed by Litwin (1980), is to allow a hash file to expand and shrink its number of buckets dynamically *without* needing a directory. Suppose that the file starts with  $M$  buckets numbered 0, 1, ...,  $M - 1$  and uses the mod hash function  $h(K) = K \bmod M$ ; this hash function is called the **initial hash function**  $h_i$ . Overflow because of collisions is still needed and can be handled by maintaining individual overflow chains for each bucket. However, when a collision leads to an overflow record in *any* file bucket, the *first* bucket in the file—bucket 0—is split into two buckets: the original bucket 0 and a new bucket  $M$  at the end of the file. The records originally in bucket 0 are distributed between the two buckets based on a different hashing function  $h_{i+1}(K) = K \bmod 2M$ . A key property of the two hash functions  $h_i$  and  $h_{i+1}$  is that any records that hashed to bucket 0



based on  $h_i$  will hash to either bucket 0 or bucket  $M$  based on  $h_{i+1}$ ; this is necessary for linear hashing to work.

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

When  $n = M$  after being incremented, this signifies that all the original buckets have been split and the hash function  $h_{i+1}$  applies to all records in the file. At this point,  $n$  is reset to 0 (zero), and any new collisions that cause overflow lead to the use of a new hashing function  $h_{i+2}(K) = K \bmod 4M$ . In general, a sequence of hashing functions  $h_{i+j}(K) = K \bmod (2^j M)$  is used, where  $j = 0, 1, 2, \dots$ ; a new hashing function  $h_{i+j+1}$  is needed whenever all the buckets  $0, 1, \dots, (2^j M) - 1$  have been split and  $n$  is reset to 0. The search for a record with hash key value  $K$  is given by Algorithm 16.3.

Splitting can be controlled by monitoring the file load factor instead of by splitting whenever an overflow occurs. In general, the **file load factor**  $l$  can be defined as  $l = r/(bfr * N)$ , where  $r$  is the current number of file records,  $bfr$  is the maximum number of records that can fit in a bucket, and  $N$  is the current number of file buckets. Buckets that have been split can also be recombined if the load factor of the file falls below a certain threshold. Blocks are combined linearly, and  $N$  is decremented appropriately. The file load can be used to trigger both splits and combinations; in this manner the file load can be kept within a desired range. Splits can be triggered when the load exceeds a certain threshold—say, 0.9—and combinations can be triggered when the load falls below another threshold—say, 0.7. The main advantages of linear hashing are that it maintains the load factor fairly constantly while the file grows and shrinks, and it does not require a directory.<sup>15</sup>

**Algorithm 16.3.** 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);

## 16.9 Other Primary File Organizations

### 16.9.1 Files of Mixed Records

The file organizations we have studied so far assume that all records of a particular file are of the same record type. The records could be of EMPLOYEES, PROJECTS, STUDENTS, or DEPARTMENTS, but each file contains records of only one type. In most database applications, we encounter situations in which numerous types of entities are interrelated in various ways, as we saw in Chapter 7. Relationships among records in various files can be represented by **connecting fields**.<sup>16</sup> For example, a

<sup>15</sup>For details of insertion and deletion into Linear hashed files, refer to Litwin (1980) and Salzberg (1988).

<sup>16</sup>The concept of foreign keys in the relational data model (Chapter 3) and references among objects in object-oriented models (Chapter 11) are examples of connecting fields.

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. In data warehouses (see Chapter 29), the input data comes from a variety of sources and undergoes an integration initially to collect the required data into an **operational data store (ODS)**. An ODS typically contains files where records of multiple types are kept together. It is passed on to a data warehouse after **ETL** (extract, transform and load) processing operations are performed on it.

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.

## 16.9.2 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 17.3.1, when we discuss the use of the B-tree data structure for indexing. In general, any data structure that can be adapted to the characteristics of disk devices can be used as a primary file organization for record placement on disk. Recently, column-based storage of data has been proposed as a primary method for storage of relations in relational databases. We will briefly introduce it in Chapter 17 as a possible alternative storage scheme for relational databases.



## 16.10 Parallelizing Disk Access Using RAID Technology

With the exponential growth in the performance and capacity of semiconductor devices and memories, faster microprocessors with larger and larger primary memories are continually becoming available. To match this growth, it is natural to expect that secondary storage technology must also take steps to keep up with processor technology in performance and reliability.

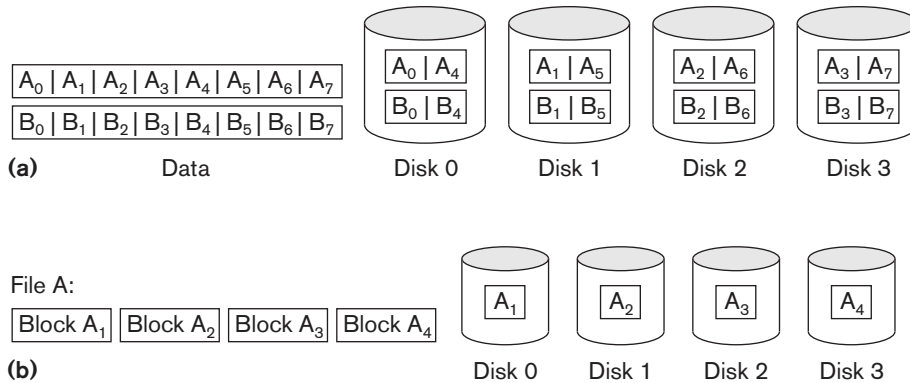
A major advance in secondary storage technology is represented by the development of **RAID**, which originally stood for **redundant arrays of inexpensive disks**. More recently, the *I* in RAID is said to stand for *independent*. The RAID idea received a very positive industry endorsement and has been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

The main goal of RAID is to even out the widely different rates of performance improvement of disks against those in memory and microprocessors.<sup>17</sup> Although RAM capacities have quadrupled every two to three years, disk *access times* are improving at less than 10% per year, and disk *transfer rates* are improving at roughly 20% per year. Disk *capacities* are indeed improving at more than 50% per year, but the speed and access time improvements are of a much smaller magnitude.

A second qualitative disparity exists between the ability of special microprocessors that cater to new applications involving video, audio, image, and spatial data processing (see Chapters 26 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 16.13 shows a file distributed or *striped* over four disks. In **bit-level striping**, a byte is split and individual bits are stored on independent disks. Figure 16.13(a) illustrates bit-striping across four disks where the bits (0, 4) are assigned to disk 0, bits (1, 5) to disk 1, and so on. With this striping, every disk participates in every read or write operation; the number of accesses per second would remain the same as on a single disk, but the amount of data read in a given time would increase fourfold. Thus, striping improves overall I/O performance by providing high overall transfer rates. **Block-level striping** stripes blocks across disks. It treats the array of disks as if it is one disk. Blocks are logically numbered from 0 in sequence. Disks in an  $m$ -disk array are numbered 0 to  $m - 1$ . With striping, block  $j$  goes to disk  $(j \bmod m)$ . Figure 16.13(b) illustrates block striping with four disks ( $m = 4$ ). Data striping also accomplishes load balancing among disks. Moreover, by storing redundant information on

<sup>17</sup>This was predicted by Gordon Bell to be about 40% every year between 1974 and 1984 and is now supposed to exceed 50% per year.



**Figure 16.13**  
Striping of data  
across multiple disks.  
(a) Bit-level striping  
across four disks.  
(b) Block-level striping  
across four disks.

disks using parity or some other error-correction code, reliability can be improved. In Sections 16.10.1 and 16.10.2, we discuss how RAID achieves the two important objectives of improved reliability and higher performance. Section 16.10.3 discusses RAID organizations and levels.

### 16.10.1 Improving Reliability with RAID

For an array of  $n$  disks, the likelihood of failure is  $n$  times as much as that for one disk. Hence, if the MTBF (mean time between failures) of a disk drive is assumed to be 200,000 hours or about 22.8 years (for the disk drive in Table 16.1 called Seagate Enterprise Performance 10K HDD, it is 1.4 million hours), the MTBF for a bank of 100 disk drives becomes only 2,000 hours or 83.3 days (for a bank of 1,000 Seagate Enterprise Performance 10K HDD disks it would be 1,400 hours or 58.33 days). Keeping a single copy of data in such an array of disks will cause a significant loss of reliability. An obvious solution is to employ redundancy of data so that disk failures can be tolerated. The disadvantages are many: additional I/O operations for write, extra computation to maintain redundancy and to do recovery from errors, and additional disk capacity to store redundant information.

One technique for introducing redundancy is called **mirroring** or **shadowing**. Data is written redundantly to two identical physical disks that are treated as one logical disk. When data is read, it can be retrieved from the disk with shorter queuing, seek, and rotational delays. If a disk fails, the other disk is used until the first is repaired. Suppose the mean time to repair is 24 hours; then the mean time to data loss of a mirrored disk system using 100 disks with MTBF of 200,000 hours each is  $(200,000)^2 / (2 * 24) = 8.33 * 10^8$  hours, which is 95,028 years.<sup>18</sup> 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.

<sup>18</sup>The formulas for MTBF calculations appear in Chen et al. (1994).

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.

### 16.10.2 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 8,192 bytes. Disk striping may be applied at a finer granularity by breaking up a byte of data into bits and spreading the bits to different disks. Thus, **bit-level data striping** consists of splitting a byte of data and writing bit  $j$  to the  $j$ th disk. With 8-bit bytes, eight physical disks may be considered as one logical disk with an eightfold increase in the data transfer rate. Each disk participates in each I/O request and the total amount of data read per request is eight times as much. Bit-level striping can be generalized to a number of disks that is either a multiple or a factor of eight. Thus, in a four-disk array, bit  $n$  goes to the disk which is  $(n \bmod 4)$ . Figure 16.13(a) shows bit-level striping of data.

The granularity of data interleaving can be higher than a bit; for example, blocks of a file can be striped across disks, giving rise to **block-level striping**. Figure 16.13(b) shows block-level data striping assuming the data file contains four blocks. With block-level striping, multiple independent requests that access single blocks (small requests) can be serviced in parallel by separate disks, thus decreasing the queuing time of I/O requests. Requests that access multiple blocks (large requests) can be parallelized, thus reducing their response time. In general, the more the number of disks in an array, the larger the potential performance benefit. However, assuming independent failures, the disk array of 100 disks collectively has 1/100th the reliability of a single disk. Thus, redundancy via error-correcting codes and disk mirroring is necessary to provide reliability along with high performance.

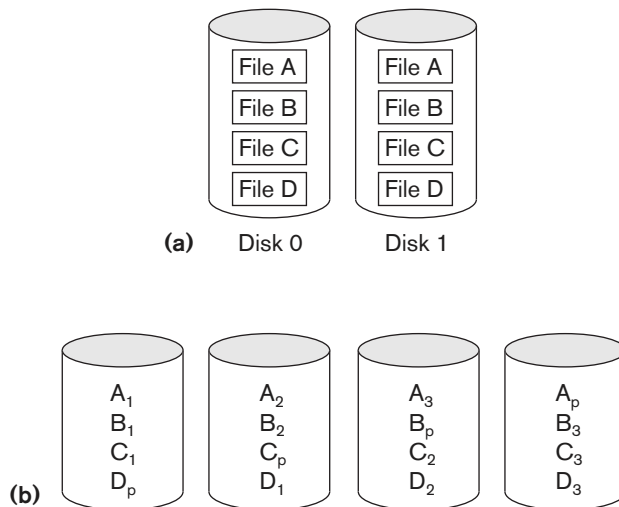
### 16.10.3 RAID Organizations and Levels

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

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

RAID level 3 uses a single parity disk relying on the disk controller to figure out which disk has failed. Levels 4 and 5 use block-level data striping, with level 5 distributing data and parity information across all disks. Figure 16.14(b) shows an illustration of RAID level 5, where parity is shown with subscript  $p$ . If one disk fails, the missing data is calculated based on the parity available from the remaining disks. Finally, RAID level 6 applies the so-called  $P + Q$  redundancy scheme using Reed-Soloman codes to protect against up to two disk failures by using just two redundant disks.

Rebuilding in case of disk failure is easiest for RAID level 1. Other levels require the reconstruction of a failed disk by reading multiple disks. Level 1 is used for critical applications such as storing logs of transactions. Levels 3 and 5 are preferred for large volume storage, with level 3 providing higher transfer rates. Most popular use of RAID technology currently uses level 0 (with striping), level 1 (with mirroring), and level 5 with an extra drive for parity. A combination of multiple RAID levels are also used—for example,  $0 + 1$  combines striping and mirroring



**Figure 16.14**

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

using a minimum of four disks. Other nonstandard RAID levels include: RAID 1.5, RAID 7, RAID-DP, RAID S or Parity RAID, Matrix RAID, RAID-K, RAID-Z, RAIDn, Linux MD RAID 10, IBM ServeRAID 1E, and unRAID. A discussion of these nonstandard levels is beyond the scope of this text. 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.

## 16.11 Modern Storage Architectures

In this section, we describe some recent developments in storage systems that are becoming an integral part of most enterprise's information system architectures. We already mentioned the SATA and SAS interface, which has almost replaced the previously popular SCSI (small computer system interface) in laptops and small servers. The Fibre Channel (FC) interface is the predominant choice for storage networks in data centers. We review some of the modern storage architectures next.

### 16.11.1 Storage Area Networks

With the rapid growth of electronic commerce, enterprise resource planning (ERP) systems that integrate application data across organizations, and data warehouses that keep historical aggregate information (see Chapter 29), the demand for storage has gone up substantially. For today's Internet-driven organizations, it has become necessary to move from a static fixed data center-oriented operation to a more flexible and dynamic infrastructure for the organizations' 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% of the overall cost of storage management. Many users of RAID systems cannot use the capacity effectively because it has to be attached in a fixed manner to one or more servers. Therefore, most large organizations have moved to a concept called **storage area networks (SANs)**. In a SAN, online storage peripherals are configured as nodes on a high-speed network and can be attached and detached from servers in a very flexible manner.

Several companies have emerged as SAN providers and supply their own proprietary topologies. They allow storage systems to be placed at longer distances from the servers and provide different performance and connectivity options. Existing storage management applications can be ported into SAN configurations using Fibre Channel networks that encapsulate the legacy SCSI protocol. As a result, the SAN-attached devices appear as SCSI devices.

Current architectural alternatives for SAN include the following: point-to-point connections between servers and storage systems via Fiber Channel; use of a Fiber

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

- Flexible many-to-many connectivity among servers and storage devices using Fiber Channel hubs and switches
- Up to 10 km separation between a server and a storage system using appropriate fiber optic cables
- Better isolation capabilities allowing nondisruptive addition of new peripherals and servers
- High-speed data replication across multiple storage systems. Typical technologies use synchronous replication for local and asynchronous replication for disaster recovery (DR) solutions.

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

### 16.11.2 Network-Attached Storage

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

NAS systems strive for reliable operation and easy administration. They include built-in features such as secure authentication, or the automatic sending of e-mail alerts in case of error on the device. The NAS devices (or *appliances*, as some vendors refer to them) are being offered with a high degree of scalability, reliability,

flexibility, and performance. Such devices typically support RAID levels 0, 1, and 5. Traditional storage area networks (SANs) differ from NAS in several ways. Specifically, SANs often utilize Fibre 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. In summary, NAS provides a file system interface with support for networked files using protocols such as common internet file system (CIFS) or network file system (NFS).

### 16.11.3 iSCSI and Other Network-Based Storage Protocols

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

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

**iSCSI storage** has mainly impacted small- and medium-sized businesses because of its combination of simplicity, low cost, and the functionality of iSCSI devices. It allows them not to learn the ins and outs of Fibre Channel (FC) technology and instead benefit from their familiarity with the IP protocol and Ethernet hardware. iSCSI implementations in the data centers of very large enterprise businesses are slow in development due to their prior investment in Fibre Channel-based SANs.

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

The latest idea to enter the enterprise IP storage race is **Fibre Channel over Ethernet (FCoE)**, which can be thought of as iSCSI without the IP. It uses many



elements of SCSI and FC (just like iSCSI), but it does not include TCP/IP components. FCoE has been successfully productized by CISCO (termed “Data Center Ethernet”) and Brocade. It takes advantage of a reliable ethernet technology that uses buffering and end-to-end flow control to avoid dropped packets. This promises excellent performance, especially on 10 Gigabit Ethernet (10GbE), and is relatively easy for vendors to add to their products.

#### 16.11.4 Automated Storage Tiering

Another trend in storage is automated storage tiering (AST), which automatically moves data between different storage types such as SATA, SAS, and solid-state drives (SSDs) depending on the need. The storage administrator can set up a tiering policy in which less frequently used data is moved to slower and cheaper SATA drives and more frequently used data is moved up to solid-state drives (see Table 16.1 for the various tiers of storage ordered by increasing speed of access). This automated tiering can improve database performance tremendously.

EMC has an implementation of this technology called FAST (fully automated storage tiering) that does continuous monitoring of data activity and takes actions to move the data to the appropriate tier based on the policy.

#### 16.11.5 Object-Based Storage

During the last few years, there have been major developments in terms of rapid growth of the cloud concept, distributed architectures for databases and for analytics, and development of data-intensive applications on the Web (see Chapters 23, 24, and 25). These developments have caused fundamental changes in enterprise storage infrastructure. The hardware-oriented file-based systems are evolving into new open-ended architectures for storage. The latest among these is object-based storage. Under this scheme, data is managed in the form of objects rather than files made of blocks. Objects carry metadata that contains properties that can be used for managing those objects. Each object carries a unique global identifier that is used to locate it. Object storage has its origins in research projects at CMU (Gibson et al., 1996) on scaling up of network attached storage and in the Oceanstore system at UC Berkeley (Kubiatowicz et al., 2000), which attempted to build a global infrastructure over all forms of trusted and untrusted servers for continuous access to persistent data. There is no need to do lower level storage operations in terms of capacity management or making decisions like what type of RAID architecture should be used for fault protection.

Object storage also allows additional flexibility in terms of interfaces—it gives control to applications that can control the objects directly and also allows the objects to be addressable across a wide namespace spanning multiple devices. Replication and distribution of objects is also supported. In general, object storage is ideally suited for scalable storage of massive amounts of unstructured data such as Web pages, images, and audio/video clips and files. Object-based storage device commands (OSDs) were proposed as part of SCSI protocol a long time ago but did not



become a commercial product until Seagate adopted OSDs in its Kinetic Open Storage Platform. Currently, Facebook uses an object storage system to store photos at the level of over 350 Petabytes of storage; Spotify uses an object storage system for storing songs; and Dropbox uses it for its storage infrastructure. Object storage is the choice of many cloud offerings, such as Amazon's AWS (Amazon Web Service) S3, and Microsoft's Azure, which stores files, relations, messages, and so on as objects. Other examples of products include Hitachi's HCP, EMC's Atmos, and Scalify's RING. Openstack Swift is an open source project that allows one to use HTTP GET and PUT to retrieve and store objects—that's basically the whole API. Openstack Swift uses very cheap hardware, is fully fault resistant, automatically takes advantage of geographic redundancy, and scales to very large numbers of objects. Since object storage forces locking to occur at the object level, it is not clear how suitable it is for concurrent transaction processing in high-throughput transaction-oriented systems. Therefore, it is still not considered viable for mainstream enterprise-level database applications.

## 16.12 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 still the preferred medium to store online database files. Table 16.1 presented a perspective on the memory hierarchies and their current capacities, access speeds, transfer rates, and costs.

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 introduced the various interface technologies in use today for disk drives and optical devices. We presented a list of strategies employed to improve access of data from disks. We also introduced solid-state drives, which are rapidly becoming popular, and optical drives, which are mainly used as tertiary storage. We discussed the working of the buffer manager, which is responsible for handling data requests and we presented various buffer replacement policies. 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, and deleted records are physically dropped 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 open addressing, chaining, or multiple hashing. Access on any nonhash field is slow, and so is ordered access of the records on any field. We discussed three hashing techniques for files that grow and shrink in the number of records dynamically: extendible, dynamic, and linear hashing. The first two use the higher-order bits of the hash address to organize a directory. Linear hashing is geared to keep the load factor of the file within a given range and adds new buckets linearly.

We briefly discussed other possibilities for primary file storage and organization, such as B-trees, and files of mixed records, which implement relationships among records of different types physically as part of the storage structure. We reviewed the recent advances in disk technology represented by RAID (redundant arrays of inexpensive (or independent) disks), which has become a standard technique in large enterprises to provide better reliability and fault tolerance features in storage. Finally, we reviewed some modern trends in enterprise storage systems: storage area networks (SANs), network-attached storage (NAS), iSCSI and other network based protocols, automatic storage tiering, and finally object-based storage, which is playing a major role in storage architecture of data centers offering cloud-based services.

## Review Questions

- 16.1.** What is the difference between primary and secondary storage?
- 16.2.** Why are disks, not tapes, used to store online database files?
- 16.3.** Define the following terms: *disk*, *disk pack*, *track*, *block*, *cylinder*, *sector*, *interblock gap*, and *read/write head*.
- 16.4.** Discuss the process of disk initialization.
- 16.5.** Discuss the mechanism used to read data from or write data to the disk.
- 16.6.** What are the components of a disk block address?

- 16.7. Why is accessing a disk block expensive? Discuss the time components involved in accessing a disk block.
- 16.8. How does double buffering improve block access time?
- 16.9. What are the reasons for having variable-length records? What types of separator characters are needed for each?
- 16.10. Discuss the techniques for allocating file blocks on disk.
- 16.11. What is the difference between a file organization and an access method?
- 16.12. What is the difference between static and dynamic files?
- 16.13. What are the typical record-at-a-time operations for accessing a file? Which of these depend on the current file record?
- 16.14. Discuss the techniques for record deletion.
- 16.15. Discuss the advantages and disadvantages of using (a) an unordered file, (b) an ordered file, and (c) a static hash file with buckets and chaining. Which operations can be performed efficiently on each of these organizations, and which operations are expensive?
- 16.16. Discuss the techniques for allowing a hash file to expand and shrink dynamically. What are the advantages and disadvantages of each?
- 16.17. What is the difference between the directories of extendible and dynamic hashing?
- 16.18. What are mixed files used for? What are other types of primary file organizations?
- 16.19. Describe the mismatch between processor and disk technologies.
- 16.20. What are the main goals of the RAID technology? How does it achieve them?
- 16.21. How does disk mirroring help improve reliability? Give a quantitative example.
- 16.22. What characterizes the levels in RAID organization?
- 16.23. What are the highlights of the popular RAID levels 0, 1, and 5?
- 16.24. What are storage area networks? What flexibility and advantages do they offer?
- 16.25. Describe the main features of network-attached storage as an enterprise storage solution.
- 16.26. How have new iSCSI systems improved the applicability of storage area networks?
- 16.27. What are SATA, SAS, and FC protocols?
- 16.28. What are solid-state drives (SSDs) and what advantage do they offer over HDDs?

- 16.29. What is the function of a buffer manager? What does it do to serve a request for data?
- 16.30. What are some of the commonly used buffer replacement strategies?
- 16.31. What are optical and tape jukeboxes? What are the different types of optical media served by optical drives?
- 16.32. What is automatic storage tiering? Why is it useful?
- 16.33. What is object-based storage? How is it superior to conventional storage systems?

## Exercises

- 16.34. Consider a disk with the following characteristics (these are not parameters of any particular disk unit): block size  $B = 512$  bytes; interblock gap size  $G = 128$  bytes; number of blocks per track = 20; number of tracks per surface = 400. A disk pack consists of 15 double-sided disks.
  - a. What is the total capacity of a track, and what is its useful capacity (excluding interblock gaps)?
  - b. How many cylinders are there?
  - c. What are the total capacity and the useful capacity of a cylinder?
  - d. What are the total capacity and the useful capacity of a disk pack?
  - e. Suppose that the disk drive rotates the disk pack at a speed of 2,400 rpm (revolutions per minute); what are the transfer rate ( $tr$ ) in bytes/msec and the block transfer time ( $btt$ ) in msec? What is the average rotational delay ( $rd$ ) in msec? What is the bulk transfer rate? (See Appendix B.)
  - f. Suppose that the average seek time is 30 msec. How much time does it take (on the average) in msec to locate and transfer a single block, given its block address?
  - g. Calculate the average time it would take to transfer 20 random blocks, and compare this with the time it would take to transfer 20 consecutive blocks using double buffering to save seek time and rotational delay.
- 16.35. A file has  $r = 20,000$  STUDENT records of *fixed length*. Each record has the following fields: Name (30 bytes), Ssn (9 bytes), Address (40 bytes), PHONE (10 bytes), Birth\_date (8 bytes), Sex (1 byte), Major\_dept\_code (4 bytes), Minor\_dept\_code (4 bytes), Class\_code (4 bytes, integer), and Degree\_program (3 bytes). An additional byte is used as a deletion marker. The file is stored on the disk whose parameters are given in Exercise 16.27.
  - a. Calculate the record size  $R$  in bytes.
  - b. Calculate the blocking factor  $bfr$  and the number of file blocks  $b$ , assuming an unspanned organization.

- c. Calculate the average time it takes to find a record by doing a linear search on the file if (i) the file blocks are stored contiguously, and double buffering is used; (ii) the file blocks are not stored contiguously.
  - d. Assume that the file is ordered by Ssn; by doing a binary search, calculate the time it takes to search for a record given its Ssn value.
- 16.36.** Suppose that only 80% of the STUDENT records from Exercise 16.28 have a value for Phone, 85% for Major\_dept\_code, 15% for Minor\_dept\_code, and 90% for Degree\_program; and suppose that we use a variable-length record file. Each record has a 1-byte *field type* for each field in the record, plus the 1-byte deletion marker and a 1-byte end-of-record marker. Suppose that we use a *spanned* record organization, where each block has a 5-byte pointer to the next block (this space is not used for record storage).
- a. Calculate the average record length  $R$  in bytes.
  - b. Calculate the number of blocks needed for the file.
- 16.37.** Suppose that a disk unit has the following parameters: seek time  $s = 20$  msec; rotational delay  $rd = 10$  msec; block transfer time  $btt = 1$  msec; block size  $B = 2400$  bytes; interblock gap size  $G = 600$  bytes. An EMPLOYEE file has the following fields: Ssn, 9 bytes; Last\_name, 20 bytes; First\_name, 20 bytes; Middle\_init, 1 byte; Birth\_date, 10 bytes; Address, 35 bytes; Phone, 12 bytes; Supervisor\_ssn, 9 bytes; Department, 4 bytes; Job\_code, 4 bytes; deletion marker, 1 byte. The EMPLOYEE file has  $r = 30,000$  records, fixed-length format, and unspanned blocking. Write appropriate formulas *and* calculate the following values for the above EMPLOYEE file:
- a. Calculate the record size  $R$  (including the deletion marker), the blocking factor  $bfr$ , and the number of disk blocks  $b$ .
  - b. Calculate the wasted space in each disk block because of the unspanned organization.
  - c. Calculate the transfer rate  $tr$  and the bulk transfer rate  $btr$  for this disk unit (see Appendix B for definitions of  $tr$  and  $btr$ ).
  - d. Calculate the average *number of block accesses* needed to search for an arbitrary record in the file, using linear search.
  - e. Calculate in msec the average *time* needed to search for an arbitrary record in the file, using linear search, if the file blocks are stored on consecutive disk blocks and double buffering is used.
  - f. Calculate in msec the average *time* needed to search for an arbitrary record in the file, using linear search, if the file blocks are *not* stored on consecutive disk blocks.
  - g. Assume that the records are ordered via some key field. Calculate the average *number of block accesses* and the *average time* needed to search for an arbitrary record in the file, using binary search.
- 16.38.** A PARTS file with Part# as the hash key includes records with the following Part# values: 2369, 3760, 4692, 4871, 5659, 1821, 1074, 7115, 1620, 2428,

3943, 4750, 6975, 4981, and 9208. The file uses eight buckets, numbered 0 to 7. Each bucket is one disk block and holds two records. Load these records into the file in the given order, using the hash function  $h(K) = K \bmod 8$ . Calculate the average number of block accesses for a random retrieval on Part#.

- 16.39. Load the records of Exercise 16.31 into expandable hash files based on extendible hashing. Show the structure of the directory at each step, and the global and local depths. Use the hash function  $h(K) = K \bmod 128$ .
- 16.40. Load the records of Exercise 16.31 into an expandable hash file, using linear hashing. Start with a single disk block, using the hash function  $h_0 = K \bmod 2^0$ , and show how the file grows and how the hash functions change as the records are inserted. Assume that blocks are split whenever an overflow occurs, and show the value of  $n$  at each stage.
- 16.41. Compare the file commands listed in Section 16.5 to those available on a file access method you are familiar with.
- 16.42. Suppose that we have an unordered file of fixed-length records that uses an unspanned record organization. Outline algorithms for insertion, deletion, and modification of a file record. State any assumptions you make.
- 16.43. Suppose that we have an ordered file of fixed-length records and an unordered overflow file to handle insertion. Both files use unspanned records. Outline algorithms for insertion, deletion, and modification of a file record and for reorganizing the file. State any assumptions you make.
- 16.44. Can you think of techniques other than an unordered overflow file that can be used to make insertions in an ordered file more efficient?
- 16.45. Suppose that we have a hash file of fixed-length records, and suppose that overflow is handled by chaining. Outline algorithms for insertion, deletion, and modification of a file record. State any assumptions you make.
- 16.46. Can you think of techniques other than chaining to handle bucket overflow in external hashing?
- 16.47. Write pseudocode for the insertion algorithms for linear hashing and for extendible hashing.
- 16.48. Write program code to access individual fields of records under each of the following circumstances. For each case, state the assumptions you make concerning pointers, separator characters, and so on. Determine the type of information needed in the file header in order for your code to be general in each case.
  - a. Fixed-length records with unspanned blocking
  - b. Fixed-length records with spanned blocking
  - c. Variable-length records with variable-length fields and spanned blocking
  - d. Variable-length records with repeating groups and spanned blocking
  - e. Variable-length records with optional fields and spanned blocking
  - f. Variable-length records that allow all three cases in parts c, d, and e

- 16.49.** Suppose that a file initially contains  $r = 120,000$  records of  $R = 200$  bytes each in an unsorted (heap) file. The block size  $B = 2,400$  bytes, the average seek time  $s = 16$  ms, the average rotational latency  $rd = 8.3$  ms, and the block transfer time  $btt = 0.8$  ms. Assume that 1 record is deleted for every 2 records added until the total number of active records is 240,000.
- How many block transfers are needed to reorganize the file?
  - How long does it take to find a record right before reorganization?
  - How long does it take to find a record right after reorganization?
- 16.50.** Suppose we have a sequential (ordered) file of 100,000 records where each record is 240 bytes. Assume that  $B = 2,400$  bytes,  $s = 16$  ms,  $rd = 8.3$  ms, and  $btt = 0.8$  ms. Suppose we want to make  $X$  independent random record reads from the file. We could make  $X$  random block reads or we could perform one exhaustive read of the entire file looking for those  $X$  records. The question is to decide when it would be more efficient to perform one exhaustive read of the entire file than to perform  $X$  individual random reads. That is, what is the value for  $X$  when an exhaustive read of the file is more efficient than random  $X$  reads? Develop this as a function of  $X$ .
- 16.51.** Suppose that a static hash file initially has 600 buckets in the primary area and that records are inserted that create an overflow area of 600 buckets. If we reorganize the hash file, we can assume that most of the overflow is eliminated. If the cost of reorganizing the file is the cost of the bucket transfers (reading and writing all of the buckets) and the only periodic file operation is the fetch operation, then how many times would we have to perform a fetch (successfully) to make the reorganization cost effective? That is, the reorganization cost and subsequent search cost are less than the search cost before reorganization. Support your answer. Assume  $s = 16$  msec,  $rd = 8.3$  msec, and  $btt = 1$  msec.
- 16.52.** Suppose we want to create a linear hash file with a file load factor of 0.7 and a blocking factor of 20 records per bucket, which is to contain 112,000 records initially.
- How many buckets should we allocate in the primary area?
  - What should be the number of bits used for bucket addresses?

## Selected Bibliography

Wiederhold (1987) has a detailed discussion and analysis of secondary storage devices and file organizations as a part of database design. Optical disks are described in Berg and Roth (1989) and analyzed in Ford and Christodoulakis (1991). Flash memory is discussed by Dipert and Levy (1993). Ruemmler and Wilkes (1994) present a survey of the magnetic-disk technology. Most textbooks on databases include discussions of the material presented here. Most data structures textbooks, including Knuth (1998), discuss static hashing in more detail; Knuth has



a complete discussion of hash functions and collision resolution techniques, as well as of their performance comparison. Knuth also offers a detailed discussion of techniques for sorting external files. Textbooks on file structures include Claybrook (1992), Smith and Barnes (1987), and Salzberg (1988); they discuss additional file organizations including tree-structured files, and have detailed algorithms for operations on files. Salzberg et al. (1990) describe a distributed external sorting algorithm. File organizations with a high degree of fault tolerance are described by Bitton and Gray (1988) and by Gray et al. (1990). Disk striping was proposed in Salem and Garcia Molina (1986). The first paper on redundant arrays of inexpensive disks (RAID) is by Patterson et al. (1988). Chen and Patterson (1990) and the excellent survey of RAID by Chen et al. (1994) are additional references. Grochowski and Hoyt (1996) discuss future trends in disk drives. Various formulas for the RAID architecture appear in Chen et al. (1994).

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

Gibson et al. (1997) describe a file server scaling approach for network-attached storage, and Kubiawicz et al. (2000) describe the Oceanstore system for creating a global utility infrastructure for storing persistent data. Both are considered pioneering approaches that led to the ideas for object-based storage. Mesnier et al. (2003) give an overview of the object storage concept. The Lustre system (Braam & Schwan, 2002) was one of the first object storage products and is used in the majority of supercomputers, including the top two, namely China's Tianhe-2 and Oakridge National Lab's Titan.

Details of disk storage devices can be found at manufacturer sites (for example, <http://www.seagate.com>, <http://www.ibm.com>, <http://www.emc.com>, <http://www.hp.com>, <http://www.storagetek.com>). IBM has a storage technology research center at IBM Almaden (<http://www.almaden.ibm.com>). Additional useful sites include CISCO storage solutions at [cisco.com](http://www.cisco.com); Network Appliance (NetApp) at [www.netapp.com](http://www.netapp.com); Hitachi Data Storage (HDS) at [www.hds.com](http://www.hds.com), and SNIA (Storage Networking Industry Association) at [www.snia.org](http://www.snia.org). A number of industry white papers are available at the aforementioned sites.