econdary
place to
transfer
a at all is
database
*blocks*, or
blocks are
*fer*. Thus,
ta so that
the same

ry storage,
o read, we
ween main
tive bytes.
ample, if a
that when
be executed

e *volatile* or
power goes
its contents
e is a power
haracteristic
presence of

ce of power.
nvolatile. On
tain types of
eir data after
S comes from
l final until it

at is typically
memory. The
main memory
units of disk

**Moore's Law**

Gordon Moore observed many years ago that integrated circuits were improving in many ways, following an exponential curve that doubles about every 18 months. Some of these parameters that follow "Moore's law" are:

1. The number of instructions per second that can be executed for unit cost. Until about 2005, the improvement was achieved by making processor chips faster, while keeping the cost fixed. After that year, the improvement has been maintained by putting progressively more processors on a single, fixed-cost chip.

2. The number of memory bits that can be bought for unit cost and the number of bits that can be put on one chip.

3. The number of bytes per unit cost on a disk and the capacity of the largest disks.

On the other hand, there are some other important parameters that do not follow Moore's law; they grow slowly if at all. Among these slowly growing parameters are the speed of accessing data in main memory and the speed at which disks rotate. Because they grow slowly, "latency" becomes progressively larger. That is, the time to move data between levels of the memory hierarchy appears enormous today, and will only get worse.

blocks (pages). Virtual memory is an artifact of the operating system and its use of the machine's hardware, and it is not a level of the memory hierarchy.

The path in Fig. 13.1 involving virtual memory represents the treatment of conventional programs and applications. It does *not* represent the typical way data in a database is managed, since a DBMS manages the data itself. However, there is increasing interest in *main-memory database systems*, which do indeed manage their data through virtual memory, relying on the operating system to bring needed data into main memory through the paging mechanism. Main-memory database systems, like most applications, are most useful when the data is small enough to remain in main memory without being swapped out by the operating system.

## 13.1.5 Exercises for Section 13.1

**Exercise 13.1.1:** Suppose that in 2008 the typical computer has a processor chip with two processors ("cores") that each run at 3 gigahertz, has a disk of 250 gigabytes, and a main memory of 1 gigabyte. Assume that Moore's law (these factors double every 18 months) holds into the indefinite future.

a) When will terabyte main memories be common?

b) When will terahertz processor chips be common (i.e., the total number of cycles per second of all the cores on a chip will be approximately $10^{12}$?

c) When will petabyte disks be common?

d) What will be a typical configuration (processor, disk, memory) in the year 2012?

! **Exercise 13.1.2:** Commander Data, the android from the 24th century on *Star Trek: The Next Generation* once proudly announced that his processor runs at "100 teraops." While an operation and a cycle may not be the same, let us suppose they are, and that Moore's law continues to hold for the next 300 years. If so, what would Data's true processor speed be?

## 13.2 Disks

The use of secondary storage is one of the important characteristics of a DBMS, and secondary storage is almost exclusively based on magnetic disks. Thus, to motivate many of the ideas used in DBMS implementation, we must examine the operation of disks in detail.

### 13.2.1 Mechanics of Disks

The two principal moving pieces of a disk drive are shown in Fig. 13.2; they are a *disk assembly* and a *head assembly*. The disk assembly consists of one or more circular *platters* that rotate around a central spindle. The upper and lower surfaces of the platters are covered with a thin layer of magnetic material, on which bits are stored. 0's and 1's are represented by different patterns in the magnetic material. A common diameter for disk platters is 3.5 inches, although disks with diameters from an inch to several feet have been built.

The disk is organized into *tracks*, which are concentric circles on a single platter. The tracks that are at a fixed radius from the center, among all the surfaces, form one *cylinder*. Tracks occupy most of a surface, except for the region closest to the spindle, as can be seen in the top view of Fig. 13.3. The density of data is much greater along a track than radially. In 2008, a typical disk has about 100,000 tracks per inch but stores about a million bits per inch along the tracks.

Tracks are organized into *sectors*, which are segments of the circle separated by *gaps* that are not magnetized to represent either 0's or 1's.[1] The sector is an indivisible unit, as far as reading and writing the disk is concerned. It is also indivisible as far as errors are concerned. Should a portion of the magnetic layer

[1]We show each track with the same number of sectors in Fig. 13.3. However, the number of sectors per track normally varies, with the outer tracks having more sectors than inner tracks.

be corrupted
sector contai
of the total t
mentioned in
between disk

The seco
*disk heads.*
surface but
destroyed).
the magneti
to an arm,
part of the

**Example 1**

### 13.2.4 Exercises for Section 13.2

**Exercise 13.2.1:** The *Megatron 777* disk has the following characteristics:

1. There are eight surfaces, with 100,000 tracks each.
2. Tracks hold an average of 2000 sectors of 1024 bytes each.
3. 10% of each track is used for gaps.
4. The disk rotates at 6,000 rpm.
5. The time it takes the head to move $n$ tracks is $1 + 0.0003n$ milliseconds.

Answer the following questions about the Megatron 777.

a) What is the capacity of the disk?

b) What is the maximum seek time?

c) What is the maximum rotational latency?

d) If a block is 65,546 bytes (i.e., 64 sectors), what is the transfer time of a block? 65536?

! e) What is the average seek time?

f) What is the average rotational latency?

g) If tracks are located on the outer inch of a 3.5-inch-diameter surface, what is the average density of bits in the sectors of a track?

!! **Exercise 13.2.2:** Prove that if we move the head from a random cylinder to another random cylinder, the average distance we move is 1/3 of the way across the disk (neglecting edge effects due to the fact that the number of cylinders is finite).

!! **Exercise 13.2.3:** Exercise 13.2.2 assumes that we move from a random track to another random track. Suppose, however, that the number of sectors per track is proportional to the length (or radius) of the track, so the bit density is the same for all tracks. Suppose also that we need to move the head from a random *sector* to another random sector. Since the sectors tend to congregate at the outside of the disk, we might expect that the average head move would be less than 1/3 of the way across the tracks. Assuming that tracks occupy radii from 0.75 inches to 1.75 inches, calculate the average number of tracks the head travels when moving between two random sectors.

! **Exercise 13.2.4:** Suppose the Megatron 747 disk head is at cylinder 4096, i.e., 1/16th of the way across the cylinders. Suppose that the next request is for a block on a random cylinder. Calculate the average time to read this block.

! **Exercise 13.2.5:** To modify a block on disk, we must read it into main memory, perform the modification, and write it back. Assume that the modification in main memory takes less time than it does for the disk to rotate, and that the disk controller postpones other requests for disk access until the block is ready to be written back to the disk. For the Megatron 747 disk, what is the time to modify a block?

## 13.3 Accelerating Access to Secondary Storage

Just because a disk takes an average of, say, 10 milliseconds to access a block, it does not follow that an application such as a database system will get the data it requests 10 milliseconds after the request is sent to the disk controller. If there is only one disk, the disk may be busy with another access for the same process or another process. In the worst case, a request for a disk access arrives more than once every 10 milliseconds, and these requests back up indefinitely. In that case, the *scheduling latency* becomes infinite.

There are several things we can do to decrease the average time a disk access takes, and thus improve the *throughput* (number of disk accesses per second that the system can accomodate). We begin this section by arguing that the "I/O model" is the right one for measuring the time database operations take. Then, we consider a number of techniques for speeding up typical database accesses to disk:

1. Place blocks that are accessed together on the same cylinder, so we can often avoid seek time, and possibly rotational latency as well.

2. Divide the data among several smaller disks rather than one large one. Having more head assemblies that can go after blocks independently can increase the number of block accesses per unit time.

3. "Mirror" a disk: making two or more copies of the data on different disks. In addition to saving the data in case one of the disks fails, this strategy, like dividing the data among several disks, lets us access several blocks at once.

4. Use a disk-scheduling algorithm, either in the operating system, in the DBMS, or in the disk controller, to select the order in which several requested blocks will be read or written.

5. Prefetch blocks to main memory in anticipation of their later use.

### 13.3.1 The I/O Model of Computation

Let us imagine a simple computer running a DBMS and trying to serve a number of users who are performing queries and database modifications. For the moment, assume our computer has one processor, one disk controller, and because that was the fourth request to arrive. The seek time is 11 for this request, since we travel from cylinder 56,000 to 16,000, more than half way across the disk. The fifth request, at cylinder 64,000, requires a seek time of 13, and the last, at 40,000, uses seek time 7. Figure 13.8 summarizes the activity caused by first-come-first-served scheduling. The difference between the two algorithms — 14 milliseconds — may not appear significant, but recall that the number of requests in this simple example is small and the algorithms were assumed not to deviate until the fourth of the six requests. □

| Cylinder of request | Time completed |
|---|---|
| 8000 | 4.3 |
| 24000 | 13.6 |
| 56000 | 26.9 |
| 16000 | 42.2 |
| 64000 | 59.5 |
| 40000 | 70.8 |

Figure 13.8: Finishing times for block accesses using the first-come-first-served algorithm

### 13.3.6 Prefetching and Large-Scale Buffering

Our final suggestion for speeding up some secondary-memory algorithms is called *prefetching* or sometimes *double buffering.* In some applications we can predict the order in which blocks will be requested from disk. If so, then we can load them into main memory buffers before they are needed. One advantage to doing so is that we are thus better able to schedule the disk, such as by using the elevator algorithm, to reduce the average time needed to access a block. In the extreme case, where there are many access requests waiting at all times, we can make the seek time per request be very close to the minimum seek time, rather than the average seek time.

### 13.3.7 Exercises for Section 13.3

**Exercise 13.3.1:** Suppose we are scheduling I/O requests for a Megatron 747 disk, and the requests in Fig. 13.9 are made, with the head initially at track 16,000. At what time is each request serviced fully if:

a) We use the elevator algorithm (it is permissible to start moving in either direction at first).

b) We use first-come-first-served scheduling.

| Cylinder of Request | First time available |
|---|---|
| 8000 | 0 |
| 48000 | 1 |
| 4000 | 10 |
| 40000 | 20 |

Figure 13.9: Arrival times for four block-access requests

**! Exercise 13.3.2:** Suppose we use two Megatron 747 disks as mirrors of one another. However, instead of allowing reads of any block from either disk, we keep the head of the first disk in the inner half of the cylinders, and the head of the second disk in the outer half of the cylinders. Assuming read requests are on random tracks, and we never have to write:

a) What is the average rate at which this system can read blocks?

b) How does this rate compare with the average rate for mirrored Megatron 747 disks with no restriction?

c) What disadvantages do you foresee for this system?

**! Exercise 13.3.3:** Let us explore the relationship between the arrival rate of requests, the throughput of the elevator algorithm, and the average delay of requests. To simplify the problem, we shall make the following assumptions:

1. A pass of the elevator algorithm always proceeds from the innermost to outermost track, or vice-versa, even if there are no requests at the extreme cylinders.

2. When a pass starts, only those requests that are already pending will be honored, not requests that come in while the pass is in progress, even if the head passes their cylinder.[2]

3. There will never be two requests for blocks on the same cylinder waiting on one pass.

Let $A$ be the interarrival rate, that is the time between requests for block accesses. Assume that the system is in steady state, that is, it has been accepting and answering requests for a long time. For a Megatron 747 disk, compute as a function of $A$:

[2]The purpose of this assumption is to avoid having to deal with the fact that a typical pass of the elevator algorithm goes fast at first, as there will be few waiting requests where the head has recently been, and slows down as it moves into an area of the disk where it has not recently been. The analysis of the way request density varies during a pass is an interesting exercise in its own right.

a) The number of

b) The average ti

c) The average ti

**! Exercise 13.3.4:** I
the average how far
blocks?

**!! Exercise 13.3.5:** I
among four disks c
pose our data is divi
random. Requests
because there are d
ter 18, for example
throughput for suc

## 13.4 Disk

In this section we s
be done to mitigat

1. The most c
attempt to r
we are able t

2. A more serio
corrupted, a
how many ti

3. A related t
a sector, bu
previously
outage duri

4. The most s
becomes un

We shall discuss
discuss "stable st
or failed writes d
collectively know

a) The number of requests serviced on one pass.

b) The average time taken to perform one pass.

c) The average time a request waits for service.

! **Exercise 13.3.4:** If we read $k$ randomly chosen blocks from one cylinder, on the average how far around the cylinder must we go before we pass all of the blocks?

!! **Exercise 13.3.5:** In Example 13.5, we saw how dividing the data to be sorted among four disks could allow more than one block to be read at a time. Suppose our data is divided randomly among $n$ disks, and requests for data are also random. Requests must be executed in the order in which they are received because there are dependencies among them that must be respected (see Chapter 18, for example, for motivation for this constraint). What is the average throughput for such a system?

## 13.4 Disk Failures

In this section we shall consider the ways in which disks can fail and what can be done to mitigate these failures.

1. The most common form of failure is an *intermittent failure*, where an attempt to read or write a sector is unsuccessful, but with repeated tries we are able to read or write successfully.

2. A more serious form of failure is one in which a bit or bits are permanently corrupted, and it becomes impossible to read a sector correctly no matter how many times we try. This form of error is called *media decay*.

3. A related type of error is a *write failure*, where we attempt to write a sector, but we can neither write successfully nor can we retrieve the previously written sector. A possible cause is that there was a power outage during the writing of the sector.

4. The most serious form of disk failure is a *disk crash*, where the entire disk becomes unreadable, suddenly and permanently.

We shall discuss parity checks as a way to detect intermittent failures. We also discuss "stable storage," a technique for organizing a disk so that media decays or failed writes do not result in permanent loss. Finally, we examine techniques collectively known as "RAID" for coping with disk crashes.

**Example 13.14:** Suppose that disks 2 and 5 fail at about the same time. Consulting the matrix of Fig. 13.10, we find that the columns for these two disks differ in row 2, where disk 2 has 1 but disk 5 has 0. We may thus reconstruct disk 2 by taking the modulo-2 sum of corresponding bits of disks 1, 4, and 6, the other three disks with 1 in row 2. Notice that none of these three disks has failed. For instance, following from the situation regarding the first blocks in Fig. 13.12, we would initially have the data of Fig. 13.13 available after disks 2 and 5 failed.

If we take the modulo-2 sum of the contents of the blocks of disks 1, 4, and 6, we find that the block for disk 2 is 00001111. This block is correct as can be verified from Fig. 13.12. The situation is now as in Fig. 13.14.

| Disk | Contents |
|---|---|
| 1) | 11110000 |
| 2) | 00001111 |
| 3) | 00111000 |
| 4) | 01000001 |
| 5) | ???????? |
| 6) | 10111110 |
| 7) | 10001001 |

Figure 13.14: After recovering disk 2

Now, we see that disk 5's column in Fig. 13.10 has a 1 in the first row. We can therefore recompute disk 5 by taking the modulo-2 sum of corresponding bits from disks 1, 2, and 3, the other three disks that have 1 in the first row. For block 1, this sum is 11000111. Again, the correctness of this calculation can be confirmed by Fig. 13.12. □

### 13.4.10 Exercises for Section 13.4

**Exercise 13.4.1:** Compute the parity bit for the following bit sequences:

a) 00111010.

b) 00000001.

c) 10101100.

**Exercise 13.4.2:** We can have two parity bits associated with a string if we follow the string by one bit that is a parity bit for the odd positions and a second that is the parity bit for the even positions. For each of the strings in Exercise 13.4.1, find the two bits that serve in this way.

## Additional Observations About RAID Level 6

1. We can combine the ideas of RAID levels 5 and 6, by varying the choice of redundant disks according to the block or cylinder number. Doing so will avoid bottlenecks when writing; the scheme described in Section 13.4.9 will cause bottlenecks at the redundant disks.

2. The scheme described in Section 13.4.9 is not restricted to four data disks. The number of disks can be one less than any power of 2, say $2^k - 1$. Of these disks, $k$ are redundant, and the remaining $2^k - k - 1$ are data disks, so the redundancy grows roughly as the logarithm of the number of data disks. For any $k$, we can construct the matrix corresponding to Fig. 13.10 by writing all possible columns of $k$ 0's and 1's, except the all-0's column. The columns with a single 1 correspond to the redundant disks, and the columns with more than one 1 are the data disks.

**Exercise 13.4.3:** Suppose we use mirrored disks as in Example 13.8, the failure rate is 5% per year, and it takes 10 hours to replace a disk. What is the mean time to a disk failure involving loss of data?

!! **Exercise 13.4.4:** Suppose we use three disks as a mirrored group; i.e., all three hold identical data. If the yearly probability of failure for one disk is $F$, and it takes $H$ hours to restore a disk, what is the mean time to data loss?

**Exercise 13.4.5:** Suppose we are using a RAID level 4 scheme with four data disks and one redundant disk. As in Example 13.9 assume blocks are a single byte. Give the block of the redundant disk if the corresponding blocks of the data disks are:

a) 01010110, 11000000, 00101011, and 10111011.

b) 11110000, 11111000, 00111100, and 01000001.

! **Exercise 13.4.6:** Suppose that a disk has probability $F$ of failing in a given year, and it takes $H$ hours to replace a disk.

a) If we use mirrored disks, what is the mean time to data loss, as a function of $F$ and $H$?

b) If we use a RAID level 4 or 5 scheme, with $N$ disks, what is the mean time to data loss?

Er

There is a
of Fig. 13.
length $n$ is
*ming distan*
they differ,
distance of
If $C$ is
bits on $n$ di
a very simp
and we can
of the two
Fig. 13.10 c
have arbitr
bits determ
If the r
bits are rec
simultaneou
positions of
could be fill
to differ in
minimum c
defines the
Thus, it ca

**Exercise 13.**
suppose that
following circu

a) The con
while th

b) The con
while th

**Exercise 13.**
changed to 01
disks must be

**Exercise 13.**
and the block
10000100, resp

## Error-Correcting Codes and RAID Level 6

There is a theory that guides our selection of a suitable matrix, like that of Fig. 13.10, to determine the content of redundant disks. A *code* of length $n$ is a set of bit-vectors (called *code words*) of length $n$. The *Hamming distance* between two code words is the number of positions in which they differ, and the *minimum distance* of a code is the smallest Hamming distance of any two different code words.

If $C$ is any code of length $n$, we can require that the corresponding bits on $n$ disks have one of the sequences that are members of the code. As a very simple example, if we are using a disk and its mirror, then $n = 2$, and we can use the code $C = \{00, 11\}$. That is, the corresponding bits of the two disks must be the same. For another example, the matrix of Fig. 13.10 defines the code consisting of the 16 bit-vectors of length 7 that have arbitrary values for the first four bits and have the remaining three bits determined by the rules for the three redundant disks.

If the minimum distance of a code is $d$, then disks whose corresponding bits are required to be a vector in the code will be able to tolerate $d - 1$ simultaneous disk crashes. The reason is that, should we obscure $d - 1$ positions of a code word, and there were two different ways these positions could be filled in to make a code word, then the two code words would have to differ in at most the $d - 1$ positions. Thus, the code could not have minimum distance $d$. As an example, the matrix of Fig. 13.10 actually defines the well-known *Hamming code*, which has minimum distance 3. Thus, it can handle two disk crashes.

**Exercise 13.4.7:** Using the same RAID level 4 scheme as in Exercise 13.4.5, suppose that data disk 1 has failed. Recover the block of that disk under the following circumstances:

a) The contents of disks 2 through 4 are 01110110, 11000000, and 00101011, while the redundant disk holds 11110011.

b) The contents of disks 2 through 4 are 11110000, 11111000, and 00110011, while the redundant disk holds 10000001.

**Exercise 13.4.8:** Suppose the block on the first disk in Exercise 13.4.5 is changed to 01010101. What changes to the corresponding blocks on the other disks must be made?

**Exercise 13.4.9:** Suppose we have the RAID level 6 scheme of Example 13.13, and the blocks of the four data disks are 00110100, 11100111, 01010101, and 10000100, respectively.

a) What are the corresponding blocks of the redundant disks?

b) If the third disk's block is rewritten to be 01111111, what steps must be taken to change other disks?

**Exercise 13.4.10:** Describe the steps taken to recover from the following failures using the RAID level 6 scheme with seven disks: (a) disks 1 and 4, (b) disks 1 and 7, (c) disks 2 and 5.

## 13.5 Arranging Data on Disk

We now turn to the matter of how disks are used to store databases. A data element such as a tuple or object is represented by a *record*, which consists of consecutive bytes in some disk block. Collections such as relations are usually represented by placing the records that represent their data elements in one or more blocks. It is normal for a disk block to hold only elements of one relation, although there are organizations where blocks hold tuples of several relations. In this section, we shall cover the basic layout techniques for both records and blocks.

### 13.5.1 Fixed-Length Records

The simplest sort of record consists of fixed-length *fields*, one for each attribute of the represented tuple. Many machines allow more efficient reading and writing of main memory when data begins at an address that is a multiple of 4 or 8; some even require us to do so. Thus, it is common to begin all fields at a multiple of 4 or 8, as appropriate. Space not used by the previous field is wasted. Note that, even though records are kept in secondary, not main, memory, they are manipulated in main memory. Thus it is necessary to lay out the record so it can be moved to main memory and accessed efficiently there.

Often, the record begins with a *header*, a fixed-length region where information about the record itself is kept. For example, we may want to keep in the record:

1. A pointer to the schema for the data stored in the record. For example, a tuple's record could point to the schema for the relation to which the tuple belongs. This information helps us find the fields of the record.

2. The length of the record. This information helps us skip over records without consulting the schema.

3. Timestamps indicating the time the record was last modified, or last read. This information may be useful for implementing database transactions as will be discussed in Chapter 18.

### 13.5.3 Exercises for Section 13.5

**Exercise 13.5.1:** Suppose a record has the following fields in this order: A character string of length 23, an integer of 2 bytes, a SQL date, and a SQL time (no decimal point). How many bytes does the record take if:

a) Fields can start at any byte.

b) Fields must start at a byte that is a multiple of 8.

c) Fields must start at a byte that is a multiple of 4.

**Exercise 13.5.2:** Assume fields are as in Exercise 13.5.1, but records also have a record header consisting of two 4-byte pointers and a character. Calculate the record length for the three situations regarding field alignment (a) through (c) in Exercise 13.5.1.

**Exercise 13.5.3:** Repeat Exercise 13.5.1 for the list of fields: a real of 8 bytes, a character string of length 25, a single byte, and a SQL date.

**Exercise 13.5.4:** Repeat Exercise 13.5.3 if the records also include a header consisting of an 8-byte pointer, and ten 2-byte integers.

## 13.6 Representing Block and Record Addresses

When in main memory, the address of a block is the virtual-memory address of its first byte, and the address of a record within that block is the virtual-memory address of the first byte of that record. However, in secondary storage, the block is not part of the application's virtual-memory address space. Rather, a sequence of bytes describes the location of the block within the overall system of data accessible to the DBMS: the device ID for the disk, the cylinder number, and so on. A record can be identified by giving its block address and the offset of the first byte of the record within the block.

In this section, we shall begin with a discussion of address spaces, especially as they pertain to the common "client-server" architecture for DBMS's (see Section 9.2.4). We then discuss the options for representing addresses, and finally look at "pointer swizzling," the ways in which we can convert addresses in the data server's world to the world of the client application programs.

### 13.6.1 Addresses in Client-Server Systems

Commonly, a database system consists of a *server* process that provides data from secondary storage to one or more *client* processes that are applications using the data. The server and client processes may be on one machine, or the server and the various clients can be distributed over many machines.

The client application uses a conventional "virtual" address space, typically 32 bits, or about 4 billion different addresses. The operating system or DBMS

### 13.6.6 Exercises for Section 13.6

**Exercise 13.6.1:** If we represent physical addresses for the Megatron 747 disk by allocating a separate byte or bytes to each of the cylinder, track within a cylinder, and block within a track, how many bytes do we need? Make a reasonable assumption about the maximum number of blocks on each track; recall that the Megatron 747 has a variable number of sectors/track.

**Exercise 13.6.2:** Repeat Exercise 13.6.1 for the Megatron 777 disk described in Exercise 13.2.1

**Exercise 13.6.3:** If we wish to represent record addresses as well as block addresses, we need additional bytes. Assuming we want addresses for a single Megatron 747 disk as in Exercise 13.6.1, how many bytes would we need for record addresses if we:

a) Included the number of the byte within a block as part of the physical address.

b) Used structured addresses for records. Assume that the stored records have a 4-byte integer as a key.

! **Exercise 13.6.4:** Suppose we wish to represent the addresses of blocks on a Megatron 747 disk logically, i.e., using identifiers of $k$ bytes for some $k$. We also need to store on the disk itself a map table, as in Fig. 13.18, consisting of pairs of logical and physical addresses. The blocks used for the map table itself are not part of the database, and therefore do not have their own logical addresses in the map table. Assuming that physical addresses use the minimum possible number of bytes for physical addresses (as calculated in Exercise 13.6.1), and logical addresses likewise use the minimum possible number of bytes for logical addresses, how many blocks of 4096 bytes does the map table for the disk occupy?

**Exercise 13.6.5:** Today, IP addresses have four bytes. Suppose that block addresses for a world-wide address system consist of an IP address for the host, a device number between 1 and 10,000, and a block address on an individual device (assumed to be a Megatron 747 disk). How many bytes would block addresses require?

**Exercise 13.6.6:** In IP version 6, IP addresses are 16 bytes long. In addition, we may want to address not only blocks, but records, which may start at any byte of a block. However, devices will have their own IP address, so there will be no need to represent a device within a host, as we suggested was necessary in Exercise 13.6.5. How many bytes would be needed to represent addresses in these circumstances, again assuming devices were Megatron 747 disks?

**Exercise 13.6.**
can perform th
separately. If th
least once is $p$, f
than on demanc

! **Exercise 13.6.**
never swizzle po
times, in some a

*i.* On-deman

*ii.* Automati

*iii.* Following

*iv.* Following

Suppose that in
or are followed
swizzling, auto
average perform

! **Exercise 13.6.**
records of 200
13.19, using 2-b
records per bloc
have its pointe
pointers to it.
before the inser
there be no roo

## 13.7 Va

Until now, we h
schema, and tha
we also may wi

1. *Data iten*
MovieSta
there mig
probably
for storin
address n

2. *Repeating*
record re
objects a

**Exercise 13.6.7:** Suppose that if we swizzle all pointers automatically, we can perform the swizzling in half the time it would take to swizzle each one separately. If the probability that a pointer in main memory will be followed at least once is $p$, for what values of $p$ is it more efficient to swizzle automatically than on demand?

! **Exercise 13.6.8:** Generalize Exercise 13.6.7 to include the possibility that we never swizzle pointers. Suppose that the important actions take the following times, in some arbitrary time units:

*i.* On-demand swizzling of a pointer: 50.

*ii.* Automatic swizzling of pointers: 15 per pointer.

*iii.* Following a swizzled pointer: 1.

*iv.* Following an unswizzled pointer: 10.

Suppose that in-memory pointers are either not followed (probability $1 - p$) or are followed $k$ times (probability $p$). For what values of $k$ and $p$ do no-swizzling, automatic-swizzling, and on-demand-swizzling each offer the best average performance?

! **Exercise 13.6.9:** Suppose that we have 4096-byte blocks in which we store records of 200 bytes. The block header consists of an offset table, as in Fig. 13.19, using 2-byte pointers to records within the block. On an average day, two records per block are inserted, and one record is deleted. A deleted record must have its pointer replaced by a "tombstone," because there may be dangling pointers to it. For specificity, assume the deletion on any day always occurs before the insertions. If the block is initially empty, after how many days will there be no room to insert any more records?

## 13.7 Variable-Length Data and Records

Until now, we have made the simplifying assumptions that records have a fixed schema, and that the schema is a list of fixed-length fields. However, in practice, we also may wish to represent:

1. *Data items whose size varies.* For instance, in Fig. 13.15 we considered a `MovieStar` relation that had an address field of up to 255 bytes. While there might be some addresses that long, the vast majority of them will probably be 50 bytes or less. We could save more than half the space used for storing `MovieStar` tuples if we used only as much space as the actual address needed.

2. *Repeating fields.* If we try to represent a many-many relationship in a record representing an object, we shall have to store references to as many objects as are related to the given object.

values in each column in the same order, then we can reconstruct the relation from the column records. Alternatively, we can keep tuple ID's or integers with each value, to tell which tuple the value belongs to.

**Example 13.22:** Consider the relation

| $X$ | $Y$ |
|---|---|
| a | b |
| c | d |
| e | f |

The column for $X$ can be represented by the record $(a, c, e)$ and the column for $Y$ can be represented by the record $(b, d, f)$. If we want to indicate the tuple to which each value belongs, then we can represent the two columns by the records $\big((1,a),\ (2,c),\ (3,e)\big)$ and $\big((1,b),\ (2,d),\ (3,f)\big)$, respectively. No matter how many tuples the relation above had, the columns would be represented by variable-length records of values or repeating groups of tuple ID's and values. □

If we store relations by columns, it is often possible to compress data, the the values all have a known type. For example, an attribute `gender` in a relation might have type `CHAR(1)`, but we would use four bytes in a tuple-based record, because it is more convenient to have all components of a tuple begin at word boundaries. However, if all we are storing is a sequence of `gender` values, then it would make sense to store the column by a sequence of bits. If we did so, we would compress the data by a factor of 32.

However, in order for column-based storage to make sense, it must be the case that most queries call for examination of all, or a large fraction of the values in each of several columns. Recall our discussion in Section 10.6 of "analytic" queries, which are the common kind of queries with the desired characteristic. These "OLAP" queries may benefit from organizing the data by columns.

## 13.7.7 Exercises for Section 13.7

**Exercise 13.7.1:** A patient record consists of the following fixed-length fields: the patient's date of birth, social-security number, and patient ID, each 9 bytes long. It also has the following variable-length fields: name, address, and patient history. If pointers within a record require 8 bytes, and the record length is a 2-byte integer, how many bytes, exclusive of the space needed for the variable-length fields, are needed for the record? You may assume that no alignment of fields is required.

**Exercise 13.7.2:** Suppose records are as in Exercise 13.7.1, and the variable-length fields name, address, and history each have a length that is uniformly distributed. For the name, the range is 20–60 bytes; for address it is 40–80 bytes, and for history it is 0–2000 bytes. What is the average length of a patient record?

One might t
compressing
to read and
we need to r
result in sig
a single tup
order to dec
to read the
read only th
when the da
In orde
press and en
sion method
large amoun
block-based
pression of a
block-by-blo

**Exercise 13.7**
mented by an
cholesterol test
Show the layou

a) The repe

b) The tests
record.

**Exercise 13.7**
we add fields f
date, and a te
suppose that f
$p$.

a) Assumin
number o
all test re

b) Repeat (
to test-re

! c) Suppose
within th

## The Merits of Data Compression

One might think that with storage so cheap, there is little advantage to compressing data. However, storing data in fewer disk blocks enables us to read and write the data faster, since we use fewer disk I/O's. When we need to read entire columns, then storage by compressed columns can result in significant speedups. However, if we want to read or write only a single tuple, then column-based storage can lose. The reason is that in order to decompress and find the value for the one tuple we want, we need to read the entire column. In contrast, tuple-based storage allows us to read only the block containing the tuple. An even more extreme case is when the data is not only compressed, but encrypted.

In order to make access of single values efficient, we must both compress and encrypt on a block-by-block basis. The most efficient compression methods generally perform better when they are allowed to compress large amounts of data as a group, and they do not lend themselves to block-based decompression. However, in special cases such as the compression of a `gender` column discussed in Section 13.7.6, we can in fact do block-by-block compression that is as good as possible.

**Exercise 13.7.3:** Suppose that the patient records of Exercise 13.7.1 are augmented by an additional repeating field that represents cholesterol tests. Each cholesterol test requires 24 bytes for a date and an integer result of the test. Show the layout of patient records if:

a) The repeating tests are kept with the record itself.

b) The tests are stored on a separate block, with pointers to them in the record.

**Exercise 13.7.4:** Starting with the patient records of Exercise 13.7.1, suppose we add fields for tests and their results. Each test consists of a test name, a date, and a test result. Assume that each such test requires 100 bytes. Also, suppose that for each patient and each test a result is stored with probability $p$.

a) Assuming pointers and integers each require 8 bytes, what is the average number of bytes devoted to test results in a patient record, assuming that all test results are kept within the record itself, as a variable-length field?

b) Repeat (a), if test results are represented by pointers within the record to test-result fields kept elsewhere.

! c) Suppose we use a hybrid scheme, where room for $k$ test results are kept within the record, and additional test results are found by following a pointer to another block (or chain of blocks) where those results are kept. As a function of $p$, what value of $k$ minimizes the amount of storage used for test results?

!! d) The amount of space used by the repeating test-result fields is not the only issue. Let us suppose that the figure of merit we wish to minimize is the number of bytes used, plus a penalty of 5000 if we have to store some results on another block (and therefore will require a disk I/O for many of the test-result accesses we need to do. Under this assumption, what is the best value of $k$ as a function of $p$?

!! **Exercise 13.7.5:** An MPEG movie uses about one gigabyte per hour of play. If we carefully organized several movies on a Megatron 747 disk, how many could we deliver with only small delay (say 100 milliseconds) from one disk. Use the timing estimates of Example 13.2, but remember that you can choose how the movies are laid out on the disk.

!! **Exercise 13.7.6:** Suppose blocks have 1000 bytes available for the storage of records, and we wish to store on them fixed-length records of length $r$, where $500 < r \leq 1000$. The value of $r$ includes the record header, but a record fragment requires an additional 32 bytes for the fragment header. For what values of $r$ can we improve space utilization by spanning records?

## 13.8 Record Modifications

Insertions, deletions, and updates of records often create special problems. These problems are most severe when the records change their length, but they come up even when records and fields are all of fixed length.

### 13.8.1 Insertion

First, let us consider insertion of new records into a relation. If the records of a relation are kept in no particular order, we can just find a block with some empty space, or get a new block if there is none, and put the record there.

There is more of a problem when the tuples must be kept in some fixed order, such as sorted by their primary key (e.g., see Section 14.1.1). If we need to insert a new record, we first locate the appropriate block for that record. Suppose first that there is space in the block to put the new record. Since records must be kept in order, we may have to slide records around in the block to make space available at the proper point. If we need to slide records, then the block organization that we showed in Fig. 13.19, which we reproduce here as Fig. 13.28, is useful. Recall from our discussion in Section 13.6.2 that we may create an "offset table" in the header of each block, with pointers to the location of each record in the block. A pointer to a record from outside the block is a "structured address," that is, the block address and the location of the entry for the record in the offset table.

### 13.8.3 Update

When a fixed-length record is updated, there is no effect on the storage system, because we know it can occupy exactly the same space it did before the update. However, when a variable-length record is updated, we have all the problems associated with both insertion and deletion, except that it is never necessary to create a tombstone for the old version of the record.

If the updated record is longer than the old version, then we may need to create more space on its block. This process may involve sliding records or even the creation of an overflow block. If variable-length portions of the record are stored on another block, as in Fig. 13.25, then we may need to move elements around that block or create a new block for storing variable-length fields. Conversely, if the record shrinks because of the update, we have the same opportunities as with a deletion to recover or consolidate space.

### 13.8.4 Exercises for Section 13.8

**Exercise 13.8.1:** Relational database systems have always preferred to use fixed-length tuples if possible. Give two reasons for this preference.

## 13.9 Summary of Chapter 13

✦ *Memory Hierarchy*: A computer system uses storage components ranging over many orders of magnitude in speed, capacity, and cost per bit. From the smallest/most expensive to largest/cheapest, they are: cache, main memory, secondary memory (disk), and tertiary memory.

✦ *Disks/Secondary Storage*: Secondary storage devices are principally magnetic disks with multigigabyte capacities. Disk units have several circular platters of magnetic material, with concentric tracks to store bits. Platters rotate around a central spindle. The tracks at a given radius from the center of a platter form a cylinder.

✦ *Blocks and Sectors*: Tracks are divided into sectors, which are separated by unmagnetized gaps. Sectors are the unit of reading and writing from the disk. Blocks are logical units of storage used by an application such as a DBMS. Blocks typically consist of several sectors.

✦ *Disk Controller*: The disk controller is a processor that controls one or more disk units. It is responsible for moving the disk heads to the proper cylinder to read or write a requested track. It also may schedule competing requests for disk access and buffers the blocks to be read or written.

✦ *Disk Access Time*: The latency of a disk is the time between a request to read or write a block, and the time the access is completed. Latency is caused principally by three factors: the seek time to move the heads to