Chapter 13

Secondary Storage Management

Database systems always involve secondary storage — the disks and other devices that store large amounts of data that persists over time. This chapter summarizes what we need to know about how a typical computer system manages storage. We review the memory hierarchy of devices with progressively slower access but larger capacity. We examine disks in particular and see how the speed of data access is affected by how we organize our data on the disk. We also study mechanisms for making disks more reliable.

Then, we turn to how data is represented. We discuss the way tuples of a relation or similar records or objects are stored. Efficiency, as always, is the key issue. We cover ways to find records quickly, and how to manage insertions and deletions of records, as well as records whose sizes grow and shrink.

13.1 The Memory Hierarchy

We begin this section by examining the memory hierarchy of a computer system. We then focus on disks, by far the most common device at the "secondary-storage" level of the hierarchy. We give the rough parameters that determine the speed of access and look at the transfer of data from disks to the lower levels of the memory hierarchy.

13.1.1 The Memory Hierarchy

A typical computer system has several different components in which data may be stored. These components have data capacities ranging over at least seven orders of magnitude and also have access speeds ranging over seven or more orders of magnitude. The cost per byte of these components also varies, but more slowly, with perhaps three orders of magnitude between the cheapest and most expensive forms of storage. Not surprisingly, the devices with smallest capacity also offer the fastest access speed and have the highest cost per byte. A schematic of the memory hierarchy is shown in Fig. 13.1.

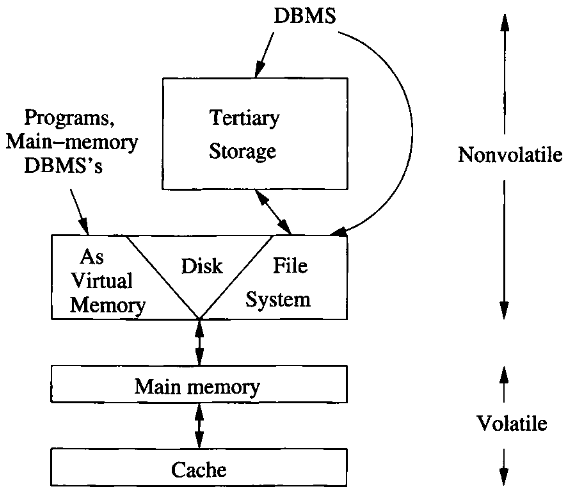

Figure 13.1: The memory hierarchy

Here are brief descriptions of the levels, from the lowest, or fastest-smallest level, up.

1. *Cache.* A typical machine has a megabyte or more of cache storage. *On-board cache* is found on the same chip as the microprocessor itself, and additional *level-2 cache* is found on another chip. Data and instructions are moved to cache from main memory when they are needed by the processor. Cached data can be accessed by the processor in a few nanoseconds.

2. *Main Memory.* In the center of the action is the computer's *main memory.* We may think of everything that happens in the computer — instruction executions and data manipulations — as working on information that is resident in main memory (although in practice, it is normal for what is used to migrate to the cache). A typical machine in 2008 is configured with about a gigabyte of main memory, although much larger main memories are possible. Typical times to move data from main memory to the processor or cache are in the 10–100 nanosecond range.

3. *Secondary Storage.* Secondary storage is typically magnetic disk, a device we shall consider in detail in Section 13.2. In 2008, single disk units have capacities of up to a terabyte, and one machine can have several disk units. The time to transfer a single byte between disk and main

Computer Quantities are Powers of 2

It is conventional to talk of sizes or capacities of computer components as if they were powers of 10: megabytes, gigabytes, and so on. In reality, since it is most efficient to design components such as memory chips to hold a number of bits that is a power of 2, all these numbers are really shorthands for nearby powers of 2. Since $2^{10} = 1024$ is very close to a thousand, we often maintain the fiction that $2^{10} = 1000$, and talk about 2^{10} with the prefix "kilo," 2^{20} as "mega," 2^{30} as "giga," 2^{40} as "tera," and 2^{50} as "peta," even though these prefixes in scientific parlance refer to 10^3, 10^6, 10^9, 10^{12} and 10^{15}, respectively. The discrepancy grows as we talk of larger numbers. A "gigabyte" is really 1.074×10^9 bytes.

We use the standard abbreviations for these numbers: K, M, G, T, and P for kilo, mega, giga, tera, and peta, respectively. Thus, 16Gb is sixteen gigabytes, or strictly speaking 2^{34} bytes. Since we sometimes want to talk about numbers that are the conventional powers of 10, we shall reserve for these the traditional numbers, without the prefixes "kilo," "mega," and so on. For example, "one million bytes" is 1,000,000 bytes, while "one megabyte" is 1,048,576 bytes.

A recent trend is to use "kilobyte," "megabyte," and so on for exact powers of ten, and to replace the third and fourth letters by "bi" to represent the similar powers of two. Thus, "kibibyte" is 1024 bytes, "mebibyte" is 1,048,576 bytes, and so on. We shall not use this convention.

memory is around 10 miliseconds. However, large numbers of bytes can be transferred at one time, so the matter of how fast data moves from and to disk is somewhat complex.

4. *Tertiary Storage.* As capacious as a collection of disk units can be, there are databases much larger than what can be stored on the disk(s) of a single machine, or even several machines. To serve such needs, ***tertiary storage*** devices have been developed to hold data volumes measured in terabytes. Tertiary storage is characterized by significantly higher read/write times than secondary storage, but also by much larger capacities and smaller cost per byte than is available from magnetic disks. Many tertiary devices involve robotic arms or conveyors that bring storage media such as magnetic tape or optical disks (e.g., DVD's) to a reading device. Retrieval takes seconds or minutes, but capacities in the petabyte range are possible.

13.1.2 Transfer of Data Between Levels

Normally, data moves between adjacent levels of the hierarchy. At the secondary and tertiary levels, accessing the desired data or finding the desired place to store data takes a great deal of time, so each level is organized to transfer large amounts of data to or from the level below, whenever any data at all is needed. Especially important for understanding the operation of a database system is the fact that the disk is organized into *disk blocks* (or just *blocks*, or as in operating systems, *pages*) of perhaps 4–64 kilobytes. Entire blocks are moved to or from a continuous section of main memory called a *buffer*. Thus, a key technique for speeding up database operations is to arrange data so that when one piece of a disk block is needed, it is likely that other data on the same block will also be needed at about the same time.

The same idea applies to other hierarchy levels. If we use tertiary storage, we try to arrange so that when we select a unit such as a DVD to read, we need much of what is on that DVD. At a lower level, movement between main memory and cache is by units of *cache lines*, typically 32 consecutive bytes. The hope is that entire cache lines will be used together. For example, if a cache line stores consecutive instructions of a program, we hope that when the first instruction is needed, the next few instructions will also be executed immediately thereafter.

13.1.3 Volatile and Nonvolatile Storage

An additional distinction among storage devices is whether they are *volatile* or *nonvolatile*. A volatile device "forgets" what is stored in it when the power goes off. A nonvolatile device, on the other hand, is expected to keep its contents intact even for long periods when the device is turned off or there is a power failure. The question of volatility is important, because one of the characteristic capabilities of a DBMS is the ability to retain its data even in the presence of errors such as power failures.

Magnetic and optical materials hold their data in the absence of power. Thus, essentially all secondary and tertiary storage devices are nonvolatile. On the other hand, main memory is generally volatile (although certain types of more expensive memory chips, such as flash memory, can hold their data after a power failure). A significant part of the complexity in a DBMS comes from the requirement that no change to the database can be considered final until it has migrated to nonvolatile, secondary storage.

13.1.4 Virtual Memory

Typical software executes in *virtual-memory*, an address space that is typically 32 bits; i.e., there are 2^{32} bytes, or 4 gigabytes, in a virtual memory. The operating system manages virtual memory, keeping some of it in main memory and the rest on disk. Transfer between memory and disk is in 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 petabyte disks be common?

b) When will terabyte main memories be common?

c) 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}?

d) What will be a typical configuration (processor, disk, memory) in the year 2015?

! **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 "12 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.

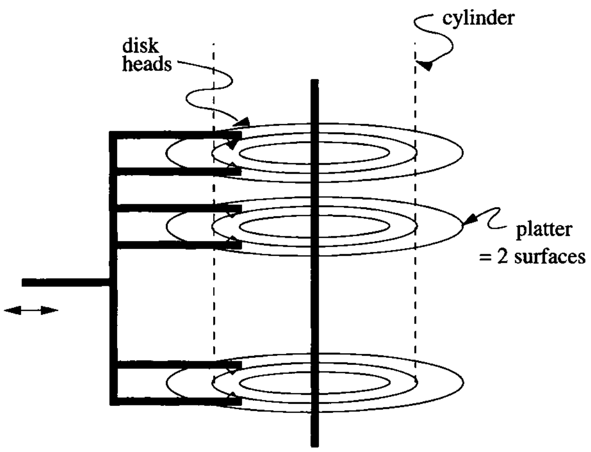

Figure 13.2: A typical disk

be corrupted in some way, so that it cannot store information, then the entire sector containing this portion cannot be used. Gaps often represent about 10% of the total track and are used to help identify the beginnings of sectors. As we mentioned in Section 13.1.2, blocks are logical units of data that are transferred between disk and main memory; blocks consist of one or more sectors.

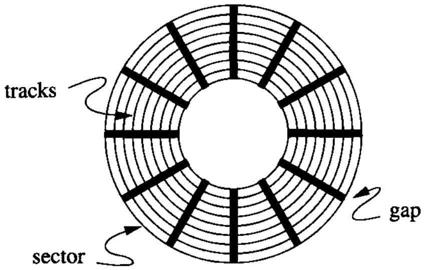

Figure 13.3: Top view of a disk surface

The second movable piece shown in Fig. 13.2, the head assembly, holds the *disk heads*. For each surface there is one head, riding extremely close to the surface but never touching it (or else a "head crash" occurs and the disk is destroyed). A head reads the magnetism passing under it, and can also alter the magnetism to write information on the disk. The heads are each attached to an arm, and the arms for all the surfaces move in and out together, being part of the rigid head assembly.

Example 13.1: The *Megatron 747* disk has the following characteristics, which are typical of a large vintage-2008 disk drive.

- There are eight platters providing sixteen surfaces.
- There are 2^{16}, or 65,536, tracks per surface.
- There are (on average) $2^8 = 256$ sectors per track.
- There are $2^{12} = 4096$ bytes per sector.

The capacity of the disk is the product of 16 surfaces, times 65,536 tracks, times 256 sectors, times 4096 bytes, or 2^{40} bytes. The Megatron 747 is thus a terabyte disk. A single track holds 256×4096 bytes, or 1 megabyte. If blocks are 2^{14}, or 16,384 bytes, then one block uses 4 consecutive sectors, and there are (on average) $256/4 = 32$ blocks on a track. □

13.2.2 The Disk Controller

One or more disk drives are controlled by a *disk controller*, which is a small processor capable of:

1. Controlling the mechanical actuator that moves the head assembly, to position the heads at a particular radius, i.e., so that any track of one particular cylinder can be read or written.
2. Selecting a sector from among all those in the cylinder at which the heads are positioned. The controller is also responsible for knowing when the rotating spindle has reached the point where the desired sector is beginning to move under the head.
3. Transferring bits between the desired sector and the computer's main memory.
4. Possibly, buffering an entire track or more in local memory of the disk controller, hoping that many sectors of this track will be read soon, and additional accesses to the disk can be avoided.

Figure 13.4 shows a simple, single-processor computer. The processor communicates via a data bus with the main memory and the disk controller. A disk controller can control several disks; we show three disks in this example.

13.2.3 Disk Access Characteristics

Accessing (reading or writing) a block requires three steps, and each step has an associated delay.

1. The disk controller positions the head assembly at the cylinder containing the track on which the block is located. The time to do so is the *seek time.*

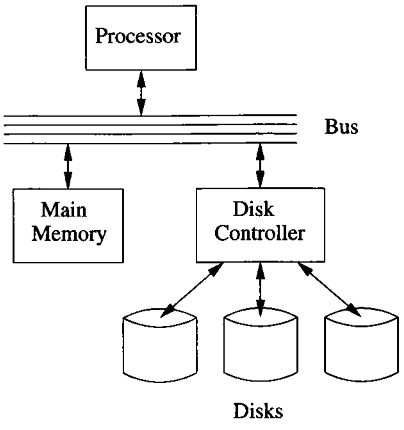

Figure 13.4: Schematic of a simple computer system

2. The disk controller waits while the first sector of the block moves under the head. This time is called the ***rotational latency.***

3. All the sectors and the gaps between them pass under the head, while the disk controller reads or writes data in these sectors. This delay is called the *transfer time.*

The sum of the seek time, rotational latency, and transfer time is the ***latency*** of the disk.

The seek time for a typical disk depends on the distance the heads have to travel from where they are currently located. If they are already at the desired cylinder, the seek time is 0. However, it takes roughly a millisecond to start the disk heads moving, and perhaps 10 milliseconds to move them across all the tracks.

A typical disk rotates once in roughly 10 milliseconds. Thus, rotational latency ranges from 0 to 10 milliseconds, and the average is 5. Transfer times tend to be much smaller, since there are often many blocks on a track. Thus, transfer times are in the sub-millisecond range. When you add all three delays, the typical average latency is about 10 milliseconds, and the maximum latency about twice that.

Example 13.2: Let us examine the time it takes to read a 16,384-byte block from the Megatron 747 disk. First, we need to know some timing properties of the disk:

- The disk rotates at 7200 rpm; i.e., it makes one rotation in 8.33 milliseconds.

- To move the head assembly between cylinders takes one millisecond to start and stop, plus one additional millisecond for every 4000 cylinders traveled. Thus, the heads move one track in 1.00025 milliseconds and move from the innermost to the outermost track, a distance of 65,536 tracks, in about 17.38 milliseconds.

- Gaps occupy 10% of the space around a track.

Let us calculate the minimum, maximum, and average times to read that 16,384-byte block. The minimum time is just the transfer time. That is, the block might be on a track over which the head is positioned already, and the first sector of the block might be about to pass under the head.

Since there are 4096 bytes per sector on the Megatron 747 (see Example 13.1 for the physical specifications of the disk), the block occupies four sectors. The heads must therefore pass over four sectors and the three gaps between them. We assume that gaps represent 10% of the circle and sectors the remaining 90%. There are 256 gaps and 256 sectors around the circle. Since the gaps together cover 36 degrees of arc and sectors the remaining 324 degrees, the total degrees of arc covered by 3 gaps and 4 sectors is $36 \times 3/256 + 324 \times 4/256 = 5.48$ degrees. The transfer time is thus $(5.48/360) \times 0.00833 = .00013$ seconds. That is, 5.48/360 is the fraction of a rotation needed to read the entire block, and .00833 seconds is the amount of time for a 360-degree rotation.

Now, let us look at the maximum possible time to read the block. In the worst case, the heads are positioned at the innermost cylinder, and the block we want to read is on the outermost cylinder (or vice versa). Thus, the first thing the controller must do is move the heads. As we observed above, the time it takes to move the Megatron 747 heads across all cylinders is about 17.38 milliseconds. This quantity is the seek time for the read.

The worst thing that can happen when the heads arrive at the correct cylinder is that the beginning of the desired block has just passed under the head. Assuming we must read the block starting at the beginning, we have to wait essentially a full rotation, or 8.33 milliseconds, for the beginning of the block to reach the head again. Once that happens, we have only to wait an amount equal to the transfer time, 0.13 milliseconds, to read the entire block. Thus, the worst-case latency is $17.38 + 8.33 + 0.13 = 25.84$ milliseconds.

Last, let us compute the average latency. Two of the components of the latency are easy to compute: the transfer time is always 0.13 milliseconds, and the average rotational latency is the time to rotate the disk half way around, or 4.17 milliseconds. We might suppose that the average seek time is just the time to move across half the tracks. However, that is not quite right, since typically, the heads are initially somewhere near the middle and therefore will have to move less than half the distance, on average, to the desired cylinder. We leave it as an exercise to show that the average distance traveled is 1/3 of the way across the disk.

The time it takes the Megatron 747 to move 1/3 of the way across the disk is $1 + (65536/3)/4000 = 6.46$ milliseconds. Our estimate of the average latency is thus $6.46+4.17+0.13 = 10.76$ milliseconds; the three terms represent average seek time, average rotational latency, and transfer time, respectively. □

13.2.4 Exercises for Section 13.2

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

1. There are ten surfaces, with 100,000 tracks each.
2. Tracks hold an average of 1000 sectors of 1024 bytes each.
3. 20% of each track is used for gaps.
4. The disk rotates at 10,000 rpm.
5. The time it takes the head to move n tracks is $1 + 0.0002n$ milliseconds.

Answer the following questions about the Megatron 777.

a) What is the capacity of the disk?

b) 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?

c) What is the maximum seek time?

d) What is the maximum rotational latency?

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

! f) What is the average seek time?

g) What is the average rotational latency?

! **Exercise 13.2.2:** Suppose the Megatron 747 disk head is at cylinder 8192, i.e., 1/8 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 13.2.3:** 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 13.2.4:** Exercise 13.2.3 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 13.2.5:** 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 one disk. The database itself is much too large to fit in main memory. Key parts of the database may be buffered in main memory, but generally, each piece of the database that one of the users accesses will have to be retrieved initially from disk. The following rule, which defines the *I/O model of computation*, can thus be assumed.

> **Dominance of I/O cost**: The time taken to perform a disk access is much larger than the time likely to be used manipulating that data in main memory. Thus, the number of block accesses (*Disk I/O's*) is a good approximation to the time needed by the algorithm and should be minimized.

Example 13.3: Suppose our database has a relation R and a query asks for the tuple of R that has a certain key value k. It is quite desirable to have an index on R to identify the disk block on which the tuple with key value k appears. However it is generally unimportant whether the index tells us where on the block this tuple appears.

For instance, if we assume a Megatron 747 disk, it will take on the order of 11 milliseconds to read a 16K-byte block. In 11 milliseconds, a modern microprocessor can execute millions of instructions. However, searching for the key value k once the block is in main memory will only take thousands of instructions, even if the dumbest possible linear search is used. The additional time to perform the search in main memory will therefore be less than 1% of the block access time and can be neglected safely. □

13.3.2 Organizing Data by Cylinders

Since seek time represents about half the time it takes to access a block, it makes sense to store data that is likely to be accessed together, such as relations, on a single cylinder, or on as many adjacent cylinders as are needed. In fact, if we choose to read all the blocks on a single track or on a cylinder consecutively, then we can neglect all but the first seek time (to move to the cylinder) and the first rotational latency (to wait until the first of the blocks moves under the head). In that case, we can approach the theoretical transfer rate for moving data on or off the disk.

Example 13.4: Suppose relation R requires 1024 blocks of a Megatron 747 disk to hold its tuples. Suppose also that we need to access all the tuples of R; for example we may be doing a search without an index or computing a sum of the values of a particular attribute of R. If the blocks holding R are distributed around the disk at random, then we shall need an average latency (10.76 milliseconds — see Example 13.2) to access each, for a total of 11 seconds.

However, 1024 blocks are exactly one cylinder of the Megatron 747. We can access them all by performing one average seek (6.46 milliseconds), after which we can read the blocks in some order, one right after another. We can read all the blocks on a cylinder in 16 rotations of the disk, since there are 16 tracks.

Sixteen rotations take $16 \times 8.33 = 133$ milliseconds. The total time to access R is thus about 139 milliseconds, and we speed up the operation on R by a factor of about 80. □

13.3.3 Using Multiple Disks

We can often improve the performance of our system if we replace one disk, with many heads locked together, by several disks with their independent heads. The arrangement was suggested in Fig. 13.4, where we showed three disks connected to a single controller. As long as the disk controller, bus, and main memory can handle n times the data-transfer rate, then n disks will have approximately the performance of one disk that operates n times as fast.

Thus, using several disks can increase the ability of a database system to handle heavy loads of disk-access requests. However, as long as the system is not overloaded (when requests will queue up and are delayed for a long time or ignored), there is no change in how long it takes to perform any single block access. If we have several disks, then the technique known as *striping* (described in the next example) will speed up access to large database objects — those that occupy a large number of blocks.

Example 13.5: Suppose we have four Megatron 747 disks and want to access the relation R of Example 13.4 faster than the 139-millisecond time that was suggested for storing R on one cylinder of one disk. We can "stripe" R by dividing it among the four disks. The first disk can receive blocks $1, 5, 9, \ldots$ of R, the second disk holds blocks $2, 6, 10, \ldots$, the third holds blocks $3, 7, 11, \ldots$, and the last disk holds blocks $4, 8, 12, \ldots$, as suggested by Fig. 13.5. Let us contrive that on each of the disks, all the blocks of R are on four tracks of a single cylinder.

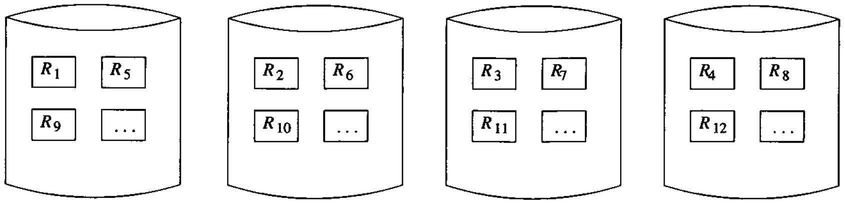

Figure 13.5: Striping a relation across four disks

Then to retrieve the 256 blocks of R on one of the disks requires an average seek time (6.46 milliseconds) plus four rotations of the disk, one rotation for each track. That is $6.46 + 4 \times 8.33 = 39.8$ milliseconds. Of course we have to wait for the last of the four disks to finish, and there is a high probability that one will take substantially more seek time than average. However, we should get a speedup in the time to access R by about a factor of three on the average, when there are four disks. □

13.3.4 Mirroring Disks

There are situations where it makes sense to have two or more disks hold identical copies of data. The disks are said to be *mirrors* of each other. One important motivation is that the data will survive a head crash by either disk, since it is still readable on a mirror of the disk that crashed. Systems designed to enhance reliability often use pairs of disks as mirrors of each other.

If we have n disks, each holding the same data, then the rate at which we can read blocks goes up by a factor of n, since the disk controller can assign a read request to any of the n disks. In fact, the speedup could be even greater than n, if a clever controller chooses to read a block from the disk whose head is currently closest to that block. Unfortunately, the writing of disk blocks does not speed up at all. The reason is that the new block must be written to each of the n disks.

13.3.5 Disk Scheduling and the Elevator Algorithm

Another effective way to improve the throughput of a disk system is to have the disk controller choose which of several requests to execute first. This approach cannot be used if accesses have to be made in a certain sequence, but if the requests are from independent processes, they can all benefit, on the average, from allowing the scheduler to choose among them judiciously.

A simple and effective way to schedule large numbers of block requests is known as the *elevator algorithm*. We think of the disk head as making sweeps across the disk, from innermost to outermost cylinder and then back again, just as an elevator makes vertical sweeps from the bottom to top of a building and back again. As heads pass a cylinder, they stop if there are one or more requests for blocks on that cylinder. All these blocks are read or written, as requested. The heads then proceed in the same direction they were traveling until the next cylinder with blocks to access is encountered. When the heads reach a position where there are no requests ahead of them in their direction of travel, they reverse direction.

Example 13.6: Suppose we are scheduling a Megatron 747 disk, which we recall has average seek, rotational latency, and transfer times of 6.46, 4.17, and 0.13, respectively (in this example, all times are in milliseconds). Suppose that at some time there are pending requests for block accesses at cylinders 8000, 24,000, and 56,000. The heads are located at cylinder 8000. In addition, there are three more requests for block accesses that come in at later times, as summarized in Fig. 13.6. For instance, the request for a block from cylinder 16,000 is made at time 10 milliseconds.

We shall assume that each block access incurs time 0.13 for transfer and 4.17 for average rotational latency, i.e., we need 4.3 milliseconds plus whatever the seek time is for each block access. The seek time can be calculated by the rule for the Megatron 747 given in Example 13.2: 1 plus the number of tracks divided by 4000. Let us see what happens if we schedule disk accesses using

Cylinder of request	First time available
8000	0
24000	0
56000	0
16000	10
64000	20
40000	30

Figure 13.6: Arrival times for four block-access requests

the elevator algorithm. The first request, at cylinder 8000, requires no seek, since the heads are already there. Thus, at time 4.3 the first access will be complete. The request for cylinder 16,000 has not arrived at this point, so we move the heads to cylinder 24,000, the next requested "stop" on our sweep to the highest-numbered tracks. The seek from cylinder 8000 to 24,000 takes 5 milliseconds, so we arrive at time 9.3 and complete the access in another 4.3. Thus, the second access is complete at time 13.6. By this time, the request for cylinder 16,000 has arrived, but we passed that cylinder at time 7.3 and will not come back to it until the next pass.

We thus move next to cylinder 56,000, taking time 9 to seek and 4.3 for rotation and transfer. The third access is thus complete at time 26.9. Now, the request for cylinder 64,000 has arrived, so we continue outward. We require 3 milliseconds for seek time, so this access is complete at time $26.9+3+4.3 = 34.2$.

At this time, the request for cylinder 40,000 has been made, so it and the request at cylinder 16,000 remain. We thus sweep inward, honoring these two requests. Figure 13.7 summarizes the times at which requests are honored.

Cylinder of request	Time completed
8000	4.3
24000	13.6
56000	26.9
64000	34.2
40000	45.5
16000	56.8

Figure 13.7: Finishing times for block accesses using the elevator algorithm

Let us compare the performance of the elevator algorithm with a more naive approach such as first-come-first-served. The first three requests are satisfied in exactly the same manner, assuming that the order of the first three requests was 8000, 24,000, and 56,000. However, at that point, we go to cylinder 16,000, 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 32,000. 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 average time taken to perform one pass.

b) The number of requests serviced on one pass.

c) The average time a request waits for service.

!! **Exercise 13.3.4:** 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?

! **Exercise 13.3.5:** 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?

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.

13.4.1 Intermittent Failures

An intermittent failure occurs if we try to read a sector, but the correct content of that sector is not delivered to the disk controller. If the controller has a way to tell that the sector is good or bad (as we shall discuss in Section 13.4.2), then the controller can reissue the read request when bad data is read, until the sector is returned correctly, or some preset limit, like 100 tries, is reached.

Similarly, the controller may attempt to write a sector, but the contents of the sector are not what was intended. The only way to check that the write was correct is to let the disk go around again and read the sector. A straightforward way to perform the check is to read the sector and compare it with the sector we intended to write. However, instead of performing the complete comparison at the disk controller, it is simpler to read the sector and see if a good sector was read. If so, we assume the write was correct, and if the sector read is bad, then the write was apparently unsuccessful and must be repeated.

13.4.2 Checksums

How a reading operation can determine the good/bad status of a sector may appear mysterious at first. Yet the technique used in modern disk drives is quite simple: each sector has some additional bits, called the *checksum*, that are set depending on the values of the data bits stored in that sector. If, on reading, we find that the checksum is not proper for the data bits, then we know there is an error in reading. If the checkum is proper, there is still a small chance that the block was not read correctly, but by using many checksum bits we can make the probability of missing a bad read arbitrarily small.

A simple form of checksum is based on the *parity* of all the bits in the sector. If there is an odd number of 1's among a collection of bits, we say the bits have *odd* parity and add a parity bit that is 1. Similarly, if there is an even number of 1's among the bits, then we say the bits have *even* parity and add parity bit 0. As a result:

- The number of 1's among a collection of bits and their parity bit is always even.

When we write a sector, the disk controller can compute the parity bit and append it to the sequence of bits written in the sector. Thus, every sector will have even parity.

Example 13.7: If the sequence of bits in a sector were 01101000, then there is an odd number of 1's, so the parity bit is 1. If we follow this sequence by its parity bit we have 011010001. If the given sequence of bits were 11101110, we have an even number of 1's, and the parity bit is 0. The sequence followed by its parity bit is 111011100. Note that each of the nine-bit sequences constructed by adding a parity bit has even parity. □

Any one-bit error in reading or writing the bits and their parity bit results in a sequence of bits that has *odd parity*; i.e., the number of 1's is odd. It is easy for the disk controller to count the number of 1's and to determine the presence of an error if a sector has odd parity.

Of course, more than one bit of the sector may be corrupted. If so, the probability is 50% that the number of 1-bits will be even, and the error will not be detected. We can increase our chances of detecting errors if we keep several parity bits. For example, we could keep eight parity bits, one for the first bit of every byte, one for the second bit of every byte, and so on, up to the eighth and last bit of every byte. Then, on a massive error, the probability is 50% that any one parity bit will detect an error, and the chance that none of the eight do so is only one in 2^8, or $1/256$. In general, if we use n independent bits as a checksum, then the chance of missing an error is only $1/2^n$. For instance, if we devote 4 bytes to a checksum, then there is only one chance in about four billion that the error will go undetected.

13.4.3 Stable Storage

While checksums will almost certainly detect the existence of a media failure or a failure to read or write correctly, it does not help us correct the error. Moreover, when writing we could find ourselves in a position where we overwrite the previous contents of a sector and yet cannot read the new contents correctly. That situation could be serious if, say, we were adding a small increment to an account balance and now have lost both the original balance and the new balance. If we could be assured that the contents of the sector contained either the new or old balance, then we would only have to determine whether the write was successful or not.

To deal with the problems above, we can implement a policy known as *stable storage* on a disk or on several disks. The general idea is that sectors are paired, and each pair represents one sector-contents X. We shall refer to the pair of sectors representing X as the "left" and "right" copies, X_L and X_R. We continue to assume that the copies are written with a sufficient number of parity-check bits so that we can rule out the possibility that a bad sector looks good when the parity checks are considered. Thus, we shall assume that if the read function returns a good value w for either X_L or X_R, then w is the true value of X. The stable-storage writing policy is:

1. Write the value of X into X_L. Check that the value has status "good"; i.e., the parity-check bits are correct in the written copy. If not, repeat the write. If after a set number of write attempts, we have not successfully written X into X_L, assume that there is a media failure in this sector. A fix-up such as substituting a spare sector for X_L must be adopted.

2. Repeat (1) for X_R.

The stable-storage reading policy is to alternate trying to read X_L and X_R, until a good value is returned. Only if no good value is returned after some large, prechosen number of tries, is X truly unreadable.

13.4.4 Error-Handling Capabilities of Stable Storage

The policies described in Section 13.4.3 are capable of compensating for several different kinds of errors. We shall outline them here.

1. *Media failures.* If, after storing X in sectors X_L and X_R, one of them undergoes a media failure and becomes permanently unreadable, we can always read X from the other. If both X_L and X_R have failed, then we cannot read X, but the probability of both failing is extremely small.

2. *Write failure.* Suppose that as we write X, there is a system failure — e.g., a power outage. It is possible that X will be lost in main memory, and also the copy of X being written at the time will be garbled. For example, half the sector may be written with part of the new value of X, while the other half remains as it was. When the system becomes available and we examine X_L and X_R, we are sure to be able to determine either the old or new value of X. The possible cases are:

   (a) The failure occurred as we were writing X_L. Then we shall find that the status of X_L is "bad." However, since we never got to write X_R, its status will be "good" (unless there is a coincident media failure at X_R, which is extremely unlikely). Thus, we can obtain the old value of X. We may also copy X_R into X_L to repair the damage to X_L.

   (b) The failure occurred after we wrote X_L. Then we expect that X_L will have status "good," and we may read the new value of X from X_L. Since X_R may or may not have the correct value of X, we should also copy X_L into X_R.

13.4.5 Recovery from Disk Crashes

The most serious mode of failure for disks is the "disk crash" or "head crash," where data is permanently destroyed. If the data was not backed up on another medium, such as a tape backup system, or on a mirror disk as we discussed in Section 13.3.4, then there is nothing we can do to recover the data. This situation represents a disaster for many DBMS applications, such as banking and other financial applications.

Several schemes have been developed to reduce the risk of data loss by disk crashes. They generally involve redundancy, extending the idea of parity checks from Section 13.4.2 or duplicated sectors, as in Section 13.4.3. The common term for this class of strategies is RAID, or *Redundant Arrays of Independent Disks.*

The rate at which disk crashes occur is generally measured by the *mean time to failure*, the time after which 50% of a population of disks can be expected to fail and be unrecoverable. For modern disks, the mean time to failure is about 10 years. We shall make the convenient assumption that if the mean time to failure is n years, then in any given year, $1/n$th of the surviving disks fail. In reality, there is a tendency for disks, like most electronic equipment, to fail early or fail late. That is, a small percentage have manufacturing defects that lead to their early demise, while those without such defects will survive for many years, until wear-and-tear causes a failure.

However, the mean time to a disk crash does not have to be the same as the mean time to data loss. The reason is that there are a number of schemes available for assuring that if one disk fails, there are others to help recover the data of the failed disk. In the remainder of this section, we shall study the most common schemes.

Each of these schemes starts with one or more disks that hold the data (we'll call these the *data disks*) and adding one or more disks that hold information that is completely determined by the contents of the data disks. The latter are called *redundant disks*. When there is a disk crash of either a data disk or a redundant disk, the other disks can be used to restore the failed disk, and there is no permanent information loss.

13.4.6 Mirroring as a Redundancy Technique

The simplest scheme is to mirror each disk, as discussed in Section 13.3.4. We shall call one of the disks the *data disk*, while the other is the *redundant disk*; which is which doesn't matter in this scheme. Mirroring, as a protection against data loss, is often referred to as *RAID level 1*. It gives a mean time to memory loss that is much greater than the mean time to disk failure, as the following example illustrates. Essentially, with mirroring and the other redundancy schemes we discuss, the only way data can be lost is if there is a second disk crash while the first crash is being repaired.

Example 13.8: Suppose each disk has a 10-year mean time to failure, which we shall take to mean that the probability of failure in any given year is 10%. If disks are mirrored, then when a disk fails, we have only to replace it with a good disk and copy the mirror disk to the new one. At the end, we have two disks that are mirrors of each other, and the system is restored to its former state.

The only thing that could go wrong is that during the copying the mirror disk fails. Now, both copies of at least part of the data have been lost, and there is no way to recover.

But how often will this sequence of events occur? Suppose that the process of replacing the failed disk takes 3 hours, which is 1/8 of a day, or 1/2920 of a year. Since we assume the average disk lasts 10 years, the probability that the mirror disk will fail during copying is $(1/10) \times (1/2920)$, or one in 29,200. If one disk fails every 10 years, then one of the two disks will fail once in 5 years on the average. One in every 29,200 of these failures results in data loss. Put another way, the mean time to a failure involving data loss is 5 × 29,200 = 146,000 years. □

13.4.7 Parity Blocks

While mirroring disks is an effective way to reduce the probability of a disk crash involving data loss, it uses as many redundant disks as there are data disks. Another approach, often called *RAID level 4*, uses only one redundant disk, no matter how many data disks there are. We assume the disks are identical, so we can number the blocks on each disk from 1 to some number n. Of course, all the blocks on all the disks have the same number of bits; for instance, the 16,384-byte blocks of the Megatron 747 have 8 × 16,384 = 131,072 bits. In the redundant disk, the ith block consists of parity checks for the ith blocks of all the data disks. That is, the jth bits of all the ith blocks, including both the data disks and the redundant disk, must have an even number of 1's among them, and we always choose the bit of the redundant disk to make this condition true.

We saw in Example 13.7 how to force the condition to be true. In the redundant disk, we choose bit j to be 1 if an odd number of the data disks have 1 in that bit, and we choose bit j of the redundant disk to be 0 if there are an even number of 1's in that bit among the data disks. The term for this calculation is the *modulo-2 sum*. That is, the modulo-2 sum of bits is 0 if there are an even number of 1's among those bits, and 1 if there are an odd number of 1's.

Example 13.9: Suppose for sake of an extremely simple example that blocks consist of only one byte — eight bits. Let there be three data disks, called 1, 2, and 3, and one redundant disk, called disk 4. Focus on the first block of all these disks. If the data disks have in their first blocks the following bit sequences:

disk 1: 11110000
disk 2: 10101010
disk 3: 00111000

then the redundant disk will have in block 1 the parity check bits:

disk 4: 01100010

Notice how in each position, an even number of the four 8-bit sequences have 1's. There are two 1's in positions 1, 2, 4, 5, and 7, four 1's in position 3, and zero 1's in positions 6 and 8. □

Reading

Reading blocks from a data disk is no different from reading blocks from any disk. There is generally no reason to read from the redundant disk, but we could.

Writing

When we write a new block of a data disk, we need not only to change that block, but we need to change the corresponding block of the redundant disk so it continues to hold the parity checks for the corresponding blocks of all the data disks. A naive approach would read the corresponding blocks of the n data disks, take their modulo-2 sum, and rewrite the block of the redundant disk. That approach requires a write of the data block that is rewritten, the reading of the $n - 1$ other data blocks, and a write of the block of the redundant disk. The total is thus $n + 1$ disk I/O's.

A better approach is to look only at the old and new versions of the data block i being rewritten. If we take their modulo-2 sum, we know in which positions there is a change in the number of 1's among the blocks numbered i on all the disks. Since these changes are always by one, any even number of 1's changes to an odd number. If we change the same positions of the redundant block, then the number of 1's in each position becomes even again. We can perform these calculations using four disk I/O's:

1. Read the old value of the data block being changed.

2. Read the corresponding block of the redundant disk.

3. Write the new data block.

4. Recalculate and write the block of the redundant disk.

Example 13.10: Suppose the three first blocks of the data disks are as in Example 13.9:

disk 1: 11110000
disk 2: 10101010
disk 3: 00111000

Suppose also that the block on the second disk changes from 10101010 to 11001100. We take the modulo-2 sum of the old and new values of the block on disk 2, to get 01100110. That tells us we must change positions 2, 3, 6, and 7 of the first block of the redundant disk. We read that block: 01100010. We replace this block by a new block that we get by changing the appropriate positions; in effect we replace the redundant block by the modulo-2 sum of itself and 01100110, to get 00000100. Another way to express the new redundant block is that it is the modulo-2 sum of the old and new versions of the block

The Algebra of Modulo-2 Sums

It may be helpful for understanding some of the tricks used with parity checks to know the algebraic rules involving the modulo-2 sum operation on bit vectors. We shall denote this operation $\oplus$. As an example, $1100 \oplus 1010 = 0110$. Here are some useful rules about $\oplus$:

- The *commutative law*: $x \oplus y = y \oplus x$.
- The *associative law*: $x \oplus (y \oplus z) = (x \oplus y) \oplus z$.
- The all-0 vector of the appropriate length, which we denote $\bar{0}$, is the *identity* for $\oplus$; that is, $x \oplus \bar{0} = \bar{0} \oplus x = x$.
- $\oplus$ is its own inverse: $x \oplus x = \bar{0}$. As a useful consequence, if $x \oplus y = z$, then we can "add" x to both sides and get $y = x \oplus z$.

being rewritten and the old value of the redundant block. In our example, the first blocks of the four disks — three data disks and one redundant — have become:

disk 1: 11110000
disk 2: 11001100
disk 3: 00111000
disk 4: 00000100

after the write to the block on the second disk and the necessary recomputation of the redundant block. Notice that in the blocks above, each column continues to have an even number of 1's. □

Failure Recovery

Now, let us consider what we would do if one of the disks crashed. If it is the redundant disk, we swap in a new disk, and recompute the redundant blocks. If the failed disk is one of the data disks, then we need to swap in a good disk and recompute its data from the other disks. The rule for recomputing any missing data is actually simple, and doesn't depend on which disk, data or redundant, is failed. Since we know that the number of 1's among corresponding bits of all disks is even, it follows that:

- The bit in any position is the modulo-2 sum of all the bits in the corresponding positions of all the other disks.

If one doubts the above rule, one has only to consider the two cases. If the bit in question is 1, then the number of corresponding bits in the other disks that are 1 must be odd, so their modulo-2 sum is 1. If the bit in question is 0, then there are an even number of 1's among the corresponding bits of the other disks, and their modulo-2 sum is 0.

Example 13.11: Suppose that disk 2 fails. We need to recompute each block of the replacement disk. Following Example 13.9, let us see how to recompute the first block of the second disk. We are given the corresponding blocks of the first and third data disks and the redundant disk, so the situation looks like:

```
disk 1: 11110000
disk 2: ????????
disk 3: 00111000
disk 4: 01100010
```

If we take the modulo-2 sum of each column, we deduce that the missing block is 10101010, as was initially the case in Example 13.9. □

13.4.8 An Improvement: RAID 5

The RAID level 4 strategy described in Section 13.4.7 effectively preserves data unless there are two almost simultaneous disk crashes. However, it suffers from a bottleneck defect that we can see when we re-examine the process of writing a new data block. Whatever scheme we use for updating the disks, we need to read and write the redundant disk's block. If there are n data disks, then the number of disk writes to the redundant disk will be n times the average number of writes to any one data disk.

However, as we observed in Example 13.11, the rule for recovery is the same as for the data disks and redundant disks: take the modulo-2 sum of corresponding bits of the other disks. Thus, we do not have to treat one disk as the redundant disk and the others as data disks. Rather, we could treat each disk as the redundant disk for some of the blocks. This improvement is often called *RAID level 5*.

For instance, if there are $n + 1$ disks numbered 0 through n, we could treat the ith cylinder of disk j as redundant if j is the remainder when i is divided by $n + 1$.

Example 13.12: In our running example, $n = 3$ so there are 4 disks. The first disk, numbered 0, is redundant for its cylinders numbered 4, 8, 12, and so on, because these are the numbers that leave remainder 0 when divided by 4. The disk numbered 1 is redundant for blocks numbered 1, 5, 9, and so on; disk 2 is redundant for blocks 2, 6, $10, \ldots$, and disk 3 is redundant for 3, 7, $11, \ldots$.

As a result, the reading and writing load for each disk is the same. If all blocks are equally likely to be written, then for one write, each disk has a 1/4 chance that the block is on that disk. If not, then it has a 1/3 chance that it will be the redundant disk for that block. Thus, each of the four disks is involved in $1/4 + (3/4) \times (1/3) = 1/2$ of the writes. □

13.4.9 Coping With Multiple Disk Crashes

There is a theory of error-correcting codes that allows us to deal with any number of disk crashes — data or redundant — if we use enough redundant disks. This strategy leads to the highest RAID "level," *RAID level 6.* We shall give only a simple example here, where two simultaneous crashes are correctable, and the strategy is based on the simplest error-correcting code, known as a *Hamming code.*

In our description we focus on a system with seven disks, numbered 1 through 7. The first four are data disks, and disks 5 through 7 are redundant. The relationship between data and redundant disks is summarized by the 3 × 7 matrix of 0's and 1's in Fig. 13.10. Notice that:

a) Every possible column of three 0's and 1's, except for the all-0 column, appears in the matrix of Fig. 13.10.

b) The columns for the redundant disks have a single 1.

c) The columns for the data disks each have at least two 1's.

	Data				Redundant		
Disk number	1	2	3	4	5	6	7
	1	1	1	0	1	0	0
	1	1	0	1	0	1	0
	1	0	1	1	0	0	1

Figure 13.10: Redundancy pattern for a system that can recover from two simultaneous disk crashes

The meaning of each of the three rows of 0's and 1's is that if we look at the corresponding bits from all seven disks, and restrict our attention to those disks that have 1 in that row, then the modulo-2 sum of these bits must be 0. Put another way, the disks with 1 in a given row of the matrix are treated as if they were the entire set of disks in a RAID level 4 scheme. Thus, we can compute the bits of one of the redundant disks by finding the row in which that disk has 1, and taking the modulo-2 sum of the corresponding bits of the other disks that have 1 in the same row.

For the matrix of Fig. 13.10, this rule implies:

1. The bits of disk 5 are the modulo-2 sum of the corresponding bits of disks 1, 2, and 3.

2. The bits of disk 6 are the modulo-2 sum of the corresponding bits of disks 1, 2, and 4.

3. The bits of disk 7 are the modulo-2 sum of the corresponding bits of disks 1, 3, and 4.

We shall see shortly that the particular choice of bits in this matrix gives us a simple rule by which we can recover from two simultaneous disk crashes.

Reading

We may read data from any data disk normally. The redundant disks can be ignored.

Writing

The idea is similar to the writing strategy outlined in Section 13.4.8, but now several redundant disks may be involved. To write a block of some data disk, we compute the modulo-2 sum of the new and old versions of that block. These bits are then added, in a modulo-2 sum, to the corresponding blocks of all those redundant disks that have 1 in a row in which the written disk also has 1.

Example 13.13: Let us again assume that blocks are only eight bits long, and focus on the first blocks of the seven disks involved in our RAID level 6 example. First, suppose the data and redundant first blocks are as given in Fig. 13.11. Notice that the block for disk 5 is the modulo-2 sum of the blocks for the first three disks, the sixth row is the modulo-2 sum of rows 1, 2, and 4, and the last row is the modulo-2 sum of rows 1, 3, and 4.

Disk	Contents
1)	11110000
2)	10101010
3)	00111000
4)	01000001
5)	01100010
6)	00011011
7)	10001001

Figure 13.11: First blocks of all disks

Suppose we rewrite the first block of disk 2 to be 00001111. If we sum this sequence of bits modulo-2 with the sequence 10101010 that is the old value of this block, we get 10100101. If we look at the column for disk 2 in Fig. 13.10, we find that this disk has 1's in the first two rows, but not the third. Since redundant disks 5 and 6 have 1 in rows 1 and 2, respectively, we must perform the sum modulo-2 operation on the current contents of their first blocks and the sequence 10100101 just calculated. That is, we flip the values of positions 1, 3, 6, and 8 of these two blocks. The resulting contents of the first blocks of all disks is shown in Fig. 13.12. Notice that the new contents continue to satisfy the constraints implied by Fig. 13.10: the modulo-2 sum of corresponding blocks that have 1 in a particular row of the matrix of Fig. 13.10 is still all 0's. □

Disk	Contents
1)	11110000
2)	00001111
3)	00111000
4)	01000001
5)	11000111
6)	10111110
7)	10001001

Figure 13.12: First blocks of all disks after rewriting disk 2 and changing the redundant disks

Failure Recovery

Now, let us see how the redundancy scheme outlined above can be used to correct up to two simultaneous disk crashes. Let the failed disks be a and b. Since all columns of the matrix of Fig. 13.10 are different, we must be able to find some row r in which the columns for a and b are different. Suppose that a has 0 in row r, while b has 1 there.

Then we can compute the correct b by taking the modulo-2 sum of corresponding bits from all the disks other than b that have 1 in row r. Note that a is not among these, so none of these disks have failed. Having recomputed b, we must recompute a, with all other disks available. Since every column of the matrix of Fig. 13.10 has a 1 in some row, we can use this row to recompute disk a by taking the modulo-2 sum of bits of those other disks with a 1 in this row.

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

Figure 13.13: Situation after disks 2 and 5 fail

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) 00111011.

b) 00000000.

c) 10101101.

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 4% per year, and it takes 8 hours to replace a disk. What is the mean time to a disk failure involving loss of data?

! **Exercise 13.4.4:** 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?

!! **Exercise 13.4.5:** 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 13.4.6: 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, 00111011, and 11111011.

b) 11110000, 11111000, 00111111, and 00000001.

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 13.4.6, 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 01010110, 11000000, and 00111011, while the redundant disk holds 11111011.

b) The contents of disks 2 through 4 are 11110000, 11111000, and 00111111, while the redundant disk holds 00000001.

Exercise 13.4.8: Suppose the block on the first disk in Exercise 13.4.6 is changed to 10101010. 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 00111100, 11000111, 01010101, and 10000100, respectively.