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.

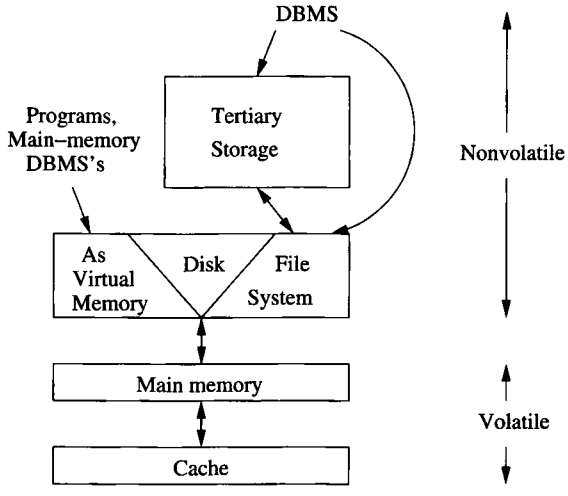

Figure 13.1: The memory hierarchy

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.

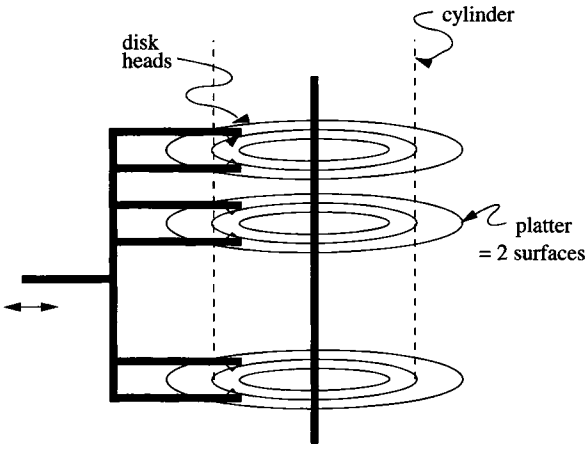

Figure 13.2: A typical disk

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.

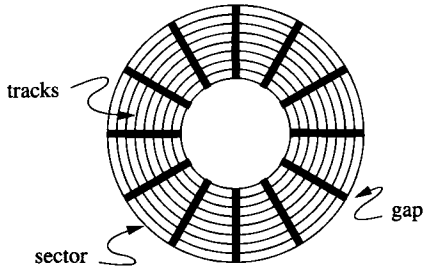

Figure 13.3: Top view of a disk surface

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*.

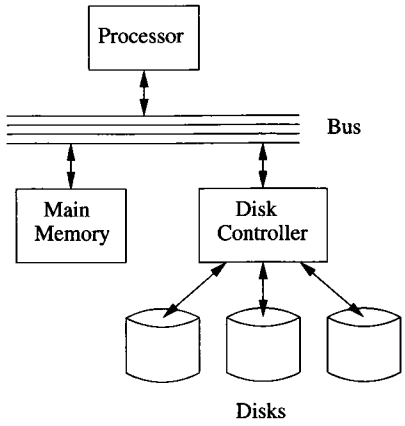

Figure 13.4: Schematic of a simple computer system

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.

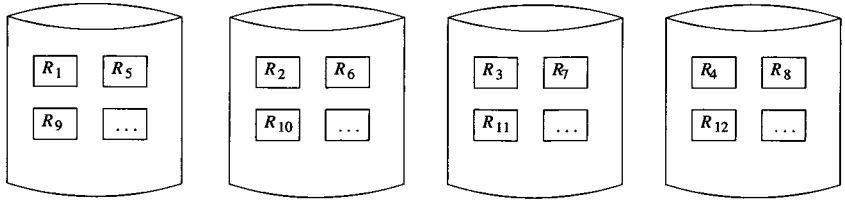

Figure 13.5: Striping a relation across four disks

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 $5 \times 29{,}200 = 146{,}000$ 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 $8 \times 16{,}384 = 131{,}072$ 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, ..., and disk 3 is redundant for 3, 7, 11, ...

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.

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

b) If the third disk's block is rewritten to be 10000000, 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 7, (b) disks 1 and 4, (c) disks 3 and 6.

13.5 Arranging Data on Disk

We now turn to the matter of how disks are used 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.

4. Pointers to the fields of the record. This information can substitute for schema information, and it will be seen to be important when we consider variable-length fields in Section 13.7.

```
CREATE TABLE MovieStar(
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    gender CHAR(1),
    birthdate DATE
);
```

Figure 13.15: A SQL table declaration

Example 13.15: Figure 13.15 repeats our running `MovieStar` schema. Let us assume all fields must start at a byte that is a multiple of four. Tuples of this relation have a header and the following four fields:

1. The first field is for `name`, and this field requires 30 bytes. If we assume that all fields begin at a multiple of 4, then we allocate 32 bytes for the `name`.

2. The next attribute is `address`. A `VARCHAR` attribute requires a fixed-length segment of bytes, with one more byte than the maximum length (for the string's endmarker). Thus, we need 256 bytes for `address`.

3. Attribute `gender` is a single byte, holding either the character `'M'` or `'F'`. We allocate 4 bytes, so the next field can start at a multiple of 4.

4. Attribute `birthdate` is a SQL `DATE` value, which is a 10-byte string. We shall allocate 12 bytes to its field, to keep subsequent records in the block aligned at multiples of 4.

. The header of the record will hold:

a) A pointer to the record schema.

b) The record length.

c) A timestamp indicating when the record was created.

We shall assume each of these items is 4 bytes long. Figure 13.16 shows the layout of a record for a `MovieStar` tuple. The length of the record is 316 bytes. □

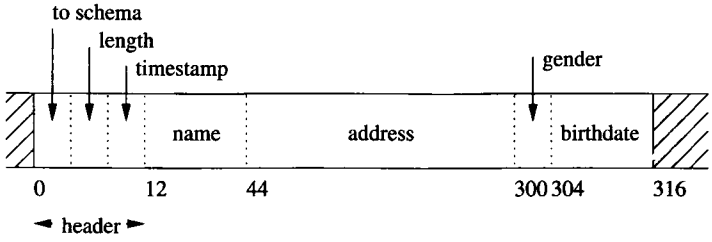

Figure 13.16: Layout of records for tuples of the `MovieStar` relation

13.5.2 Packing Fixed-Length Records into Blocks

Records representing tuples of a relation are stored in blocks of the disk and moved into main memory (along with their entire block) when we need to access or update them. The layout of a block that holds records is suggested in Fig. 13.17.

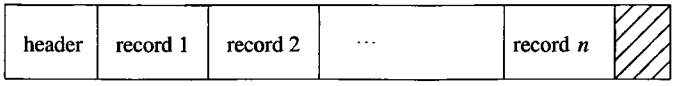

Figure 13.17: A typical block holding records

In addition to the records, there is a *block header* holding information such as:

1. Links to one or more other blocks that are part of a network of blocks such as those that will be described in Chapter 14 for creating indexes to the tuples of a relation.
2. Information about the role played by this block in such a network.
3. Information about which relation the tuples of this block belong to.
4. A "directory" giving the offset of each record in the block.
5. Timestamp(s) indicating the time of the block's last modification and/or access.

By far the simplest case is when the block holds tuples from one relation, and the records for those tuples have a fixed format. In that case, following the header, we pack as many records as we can into the block and leave the remaining space unused.

Example 13.16: Suppose we are storing records with the layout developed in Example 13.15. These records are 316 bytes long. Suppose also that we use 4096-byte blocks. Of these bytes, say 12 will be used for a block header, leaving 4084 bytes for data. In this space we can fit twelve records of the given 316-byte format, and 292 bytes of each block are wasted space. □

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 15, 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 4.

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

Exercise 13.5.2: Repeat Exercise 13.5.1 for the list of fields: a real of 8 bytes, a character string of length 17, a single byte, and a SQL date.

Exercise 13.5.3: 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 13.5.4: Repeat Exercise 13.5.2 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

decides which parts of the address space are currently located in main memory, and hardware maps the virtual address space to physical locations in main memory. We shall not think further of this virtual-to-physical translation, and shall think of the client address space as if it were main memory itself.

The server's data lives in a *database address space*. The addresses of this space refer to blocks, and possibly to offsets within the block. There are several ways that addresses in this address space can be represented:

1. *Physical Addresses.* These are byte strings that let us determine the place within the secondary storage system where the block or record can be found. One or more bytes of the physical address are used to indicate each of:

   (a) The host to which the storage is attached (if the database is stored across more than one machine),

   (b) An identifier for the disk or other device on which the block is located,

   (c) The number of the cylinder of the disk,

   (d) The number of the track within the cylinder,

   (e) The number of the block within the track, and

   (f) (In some cases) the offset of the beginning of the record within the block.

2. *Logical Addresses.* Each block or record has a "logical address," which is an arbitrary string of bytes of some fixed length. A *map table*, stored on disk in a known location, relates logical to physical addresses, as suggested in Fig. 13.18.

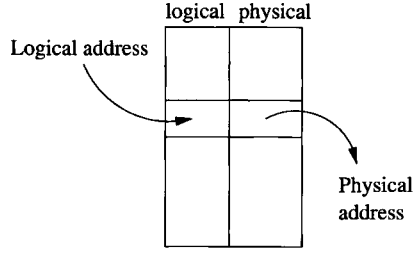

Figure 13.18: A map table translates logical to physical addresses

Notice that physical addresses are long. Eight bytes is about the minimum we could use if we incorporate all the listed elements, and some systems use many more bytes. For example, imagine a database of objects that is designed to last for 100 years. In the future, the database may grow to encompass one

million machines, and each machine might be fast enough to create one object every nanosecond. This system would create around 2^{77} objects, which requires a minimum of ten bytes to represent addresses. Since we would probably prefer to reserve some bytes to represent the host, others to represent the storage unit, and so on, a rational address notation would use considerably more than 10 bytes for a system of this scale.

13.6.2 Logical and Structured Addresses

One might wonder what the purpose of logical addresses could be. All the information needed for a physical address is found in the map table, and following logical pointers to records requires consulting the map table and then going to the physical address. However, the level of indirection involved in the map table allows us considerable flexibility. For example, many data organizations require us to move records around, either within a block or from block to block. If we use a map table, then all pointers to the record refer to this map table, and all we have to do when we move or delete the record is to change the entry for that record in the table.

Many combinations of logical and physical addresses are possible as well, yielding *structured* address schemes. For instance, one could use a physical address for the block (but not the offset within the block), and add the key value for the record being referred to. Then, to find a record given this structured address, we use the physical part to reach the block containing that record, and we examine the records of the block to find the one with the proper key.

A similar, and very useful, combination of physical and logical addresses is to keep in each block an *offset table* that holds the offsets of the records within the block, as suggested in Fig. 13.19. Notice that the table grows from the front end of the block, while the records are placed starting at the end of the block. This strategy is useful when the records need not be of equal length. Then, we do not know in advance how many records the block will hold, and we do not have to allocate a fixed amount of the block header to the table initially.

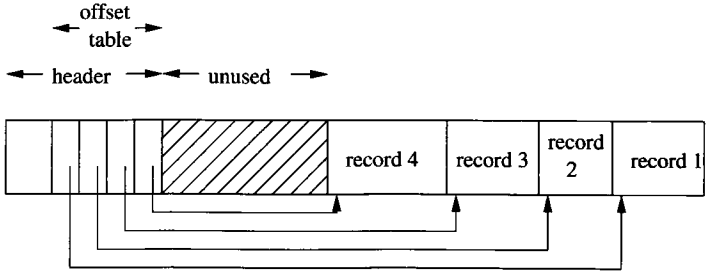

Figure 13.19: A block with a table of offsets telling us the position of each record within the block

The address of a record is now the physical address of its block plus the offset

of the entry in the block's offset table for that record. This level of indirection within the block offers many of the advantages of logical addresses, without the need for a global map table.

- We can move the record around within the block, and all we have to do is change the record's entry in the offset table; pointers to the record will still be able to find it.

- We can even allow the record to move to another block, if the offset table entries are large enough to hold a *forwarding address* for the record, giving its new location.

- Finally, we have an option, should the record be deleted, of leaving in its offset-table entry a *tombstone*, a special value that indicates the record has been deleted. Prior to its deletion, pointers to this record may have been stored at various places in the database. After record deletion, following a pointer to this record leads to the tombstone, whereupon the pointer can either be replaced by a null pointer, or the data structure otherwise modified to reflect the deletion of the record. Had we not left the tombstone, the pointer might lead to some new record, with surprising, and erroneous, results.

13.6.3 Pointer Swizzling

Often, pointers or addresses are part of records. This situation is not typical for records that represent tuples of a relation, but it is common for tuples that represent objects. Also, modern object-relational database systems allow attributes of pointer type (called references), so even relational systems need the ability to represent pointers in tuples. Finally, index structures are composed of blocks that usually have pointers within them. Thus, we need to study the management of pointers as blocks are moved between main and secondary memory.

As we mentioned earlier, every block, record, object, or other referenceable data item has two forms of address: its *database address* in the server's address space, and a *memory address* if the item is currently copied in virtual memory. When in secondary storage, we surely must use the database address of the item. However, when the item is in the main memory, we can refer to the item by either its database address or its memory address. It is more efficient to put memory addresses wherever an item has a pointer, because these pointers can be followed using a single machine instruction.

In contrast, following a database address is much more time-consuming. We need a table that translates from all those database addresses that are currently in virtual memory to their current memory address. Such a *translation table* is suggested in Fig. 13.20. It may look like the map table of Fig. 13.18 that translates between logical and physical addresses. However:

a) Logical and physical addresses are both representations for the database address. In contrast, memory addresses in the translation table are for copies of the corresponding object in memory.

b) All addressable items in the database have entries in the map table, while only those items currently in memory are mentioned in the translation table.

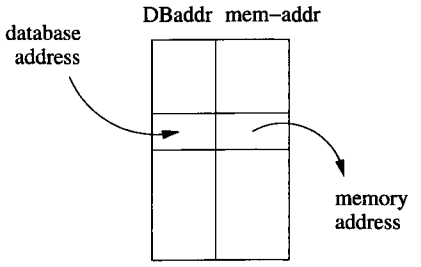

Figure 13.20: The translation table turns database addresses into their equivalents in memory

To avoid the cost of translating repeatedly from database addresses to memory addresses, several techniques have been developed that are collectively known as *pointer swizzling*. The general idea is that when we move a block from secondary to main memory, pointers within the block may be "swizzled," that is, translated from the database address space to the virtual address space. Thus, a pointer actually consists of:

1. A bit indicating whether the pointer is currently a database address or a (swizzled) memory address.

2. The database or memory pointer, as appropriate. The same space is used for whichever address form is present at the moment. Of course, not all the space may be used when the memory address is present, because it is typically shorter than the database address.

Example 13.17: Figure 13.21 shows a simple situation in which the Block 1 has a record with pointers to a second record on the same block and to a record on another block. The figure also shows what might happen when Block 1 is copied to memory. The first pointer, which points within Block 1, can be swizzled so it points directly to the memory address of the target record.

However, if Block 2 is not in memory at this time, then we cannot swizzle the second pointer; it must remain unswizzled, pointing to the database address of its target. Should Block 2 be brought to memory later, it becomes theoretically possible to swizzle the second pointer of Block 1. Depending on the swizzling strategy used, there may or may not be a list of such pointers that are in

memory, referring to Block 2; if so, then we have the option of swizzling the pointer at that time. □

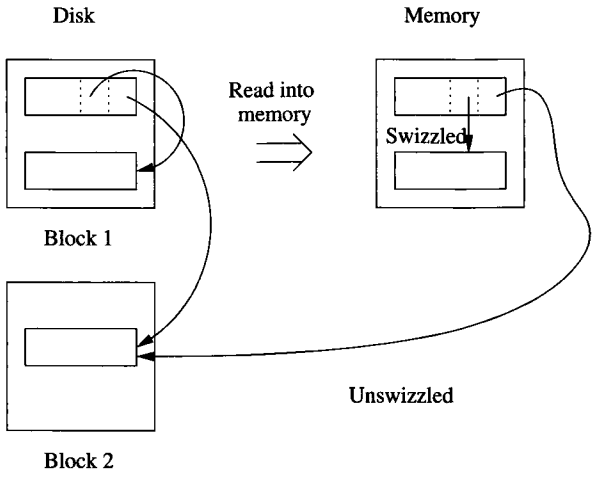

Figure 13.21: Structure of a pointer when swizzling is used

Automatic Swizzling

There are several strategies we can use to determine when to swizzle pointers. If we use *automatic swizzling*, then as soon as a block is brought into memory, we locate all its pointers and addresses and enter them into the translation table if they are not already there. These pointers include both the pointers *from* records in the block to elsewhere and the addresses of the block itself and/or its records, if these are addressable items. We need some mechanism to locate the pointers within the block. For example:

1. If the block holds records with a known schema, the schema will tell us where in the records the pointers are found.

2. If the block is used for one of the index structures we shall discuss in Chapter 14, then the block will hold pointers at known locations.

3. We may keep within the block header a list of where the pointers are.

When we enter into the translation table the addresses for the block just moved into memory, and/or its records, we know where in memory the block has been buffered. We may thus create the translation-table entry for these database addresses straightforwardly. When we insert one of these database addresses A into the translation table, we may find it in the table already, because its block is currently in memory. In this case, we replace A in the block

just moved to memory by the corresponding memory address, and we set the "swizzled" bit to true. On the other hand, if A is not yet in the translation table, then its block has not been copied into main memory. We therefore cannot swizzle this pointer and leave it in the block as a database pointer.

Suppose that during the use of this data, we follow a pointer P and we find that P is still unswizzled, i.e., in the form of a database pointer. We consult the translation table to see if database address P currently has a memory equivalent. If not, block B must be copied into a memory buffer. Once B is in memory, we can "swizzle" P by replacing its database form by the equivalent memory form.

Swizzling on Demand

Another approach is to leave all pointers unswizzled when the block is first brought into memory. We enter its address, and the addresses of its pointers, into the translation table, along with their memory equivalents. If we follow a pointer P that is inside some block of memory, we swizzle it, using the same strategy that we followed when we found an unswizzled pointer using automatic swizzling.

The difference between on-demand and automatic swizzling is that the latter tries to get all the pointers swizzled quickly and efficiently when the block is loaded into memory. The possible time saved by swizzling all of a block's pointers at one time must be weighed against the possibility that some swizzled pointers will never be followed. In that case, any time spent swizzling and unswizzling the pointer will be wasted.

An interesting option is to arrange that database pointers look like invalid memory addresses. If so, then we can allow the computer to follow any pointer as if it were in its memory form. If the pointer happens to be unswizzled, then the memory reference will cause a hardware trap. If the DBMS provides a function that is invoked by the trap, and this function "swizzles" the pointer in the manner described above, then we can follow swizzled pointers in single instructions, and only need to do something more time consuming when the pointer is unswizzled.

No Swizzling

Of course it is possible never to swizzle pointers. We still need the translation table, so the pointers may be followed in their unswizzled form. This approach does offer the advantage that records cannot be pinned in memory, as discussed in Section 13.6.5, and decisions about which form of pointer is present need not be made.

Programmer Control of Swizzling

In some applications, it may be known by the application programmer whether the pointers in a block are likely to be followed. This programmer may be able

to specify explicitly that a block loaded into memory is to have its pointers swizzled, or the programmer may call for the pointers to be swizzled only as needed. For example, if a programmer knows that a block is likely to be accessed heavily, such as the root block of a B-tree (discussed in Section 14.2), then the pointers would be swizzled. However, blocks that are loaded into memory, used once, and then likely dropped from memory, would not be swizzled.

13.6.4 Returning Blocks to Disk

When a block is moved from memory back to disk, any pointers within that block must be "unswizzled"; that is, their memory addresses must be replaced by the corresponding database addresses. The translation table can be used to associate addresses of the two types in either direction, so in principle it is possible to find, given a memory address, the database address to which the memory address is assigned.

However, we do not want each unswizzling operation to require a search of the entire translation table. While we have not discussed the implementation of this table, we might imagine that the table of Fig. 13.20 has appropriate indexes. If we think of the translation table as a relation, then the problem of finding the memory address associated with a database address x can be expressed as the query:

```
SELECT memAddr
FROM TranslationTable
WHERE dbAddr = x;
```

For instance, a hash table using the database address as the key might be appropriate for an index on the `dbAddr` attribute; Chapter 14 suggests possible data structures.

If we want to support the reverse query,

```
SELECT dbAddr
FROM TranslationTable
WHERE memAddr = y;
```

then we need to have an index on attribute `memAddr` as well. Again, Chapter 14 suggests data structures suitable for such an index. Also, Section 13.6.5 talks about linked-list structures that in some circumstances can be used to go from a memory address to all main-memory pointers to that address.

13.6.5 Pinned Records and Blocks

A block in memory is said to be *pinned* if it cannot at the moment be written back to disk safely. A bit telling whether or not a block is pinned can be located in the header of the block. There are many reasons why a block could be pinned, including requirements of a recovery system as discussed in Chapter 17. Pointer swizzling introduces an important reason why certain blocks must be pinned.

If a block B_1 has within it a swizzled pointer to some data item in block B_2, then we must be very careful about moving block B_2 back to disk and reusing its main-memory buffer. The reason is that, should we follow the pointer in B_1, it will lead us to the buffer, which no longer holds B_2; in effect, the pointer has become dangling. A block, like B_2, that is referred to by a swizzled pointer from somewhere else is therefore pinned.

When we write a block back to disk, we not only need to "unswizzle" any pointers in that block. We also need to make sure it is not pinned. If it is pinned, we must either unpin it, or let the block remain in memory, occupying space that could otherwise be used for some other block. To unpin a block that is pinned because of swizzled pointers from outside, we must "unswizzle" any pointers to it. Consequently, the translation table must record, for each database address whose data item is in memory, the places in memory where swizzled pointers to that item exist. Two possible approaches are:

1. Keep the list of references to a memory address as a linked list attached to the entry for that address in the translation table.

2. If memory addresses are significantly shorter than database addresses, we can create the linked list in the space used for the pointers themselves. That is, each space used for a database pointer is replaced by

   (a) The swizzled pointer, and

   (b) Another pointer that forms part of a linked list of all occurrences of this pointer.

   Figure 13.22 suggests how two occurrences of a memory pointer y could be linked, starting at the entry in the translation table for database address x and its corresponding memory address y.

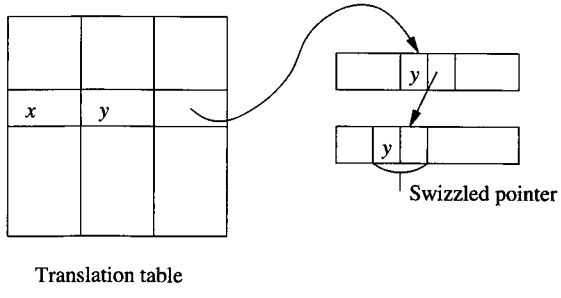

Figure 13.22: A linked list of occurrences of a swizzled pointer

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: 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 1000, and a block address on an individual device (assumed to be a Megatron 747 disk). How many bytes would block addresses require?

Exercise 13.6.5: 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 13.6.4. How many bytes would be needed to represent addresses in these circumstances, again assuming devices were Megatron 747 disks?

! **Exercise 13.6.6:** 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 13.6.7:** Suppose that we have 4096-byte blocks in which we store records of 100 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?

Exercise 13.6.8: 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 13.6.9:** Generalize Exercise 13.6.8 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: 30.

ii. Automatic swizzling of pointers: 20 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?

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.

3. *Variable-format records.* Sometimes we do not know in advance what the fields of a record will be, or how many occurrences of each field there will be. An important example is a record that represents an XML element, which might have no constraints at all, or might be allowed to have repeating subelements, optional attributes, and so on.

4. *Enormous fields.* Modern DBMS's support attributes whose values are very large. For instance, a movie record might have a field that is a 2-gigabyte MPEG encoding of the movie itself, as well as more mundane fields such as the title of the movie.

13.7.1 Records With Variable-Length Fields

If one or more fields of a record have variable length, then the record must contain enough information to let us find any field of the record. A simple but effective scheme is to put all fixed-length fields ahead of the variable-length fields. We then place in the record header:

1. The length of the record.

2. Pointers to (i.e., offsets of) the beginnings of all the variable-length fields other than the first (which we know must immediately follow the fixed-length fields).

Example 13.18: Suppose we have movie-star records with name, address, gender, and birthdate. We shall assume that the gender and birthdate are fixed-length fields, taking 4 and 12 bytes, respectively. However, both name and address will be represented by character strings of whatever length is appropriate. Figure 13.23 suggests what a typical movie-star record would look like. Note that no pointer to the beginning of the name is needed; that field begins right after the fixed-length portion of the record. □

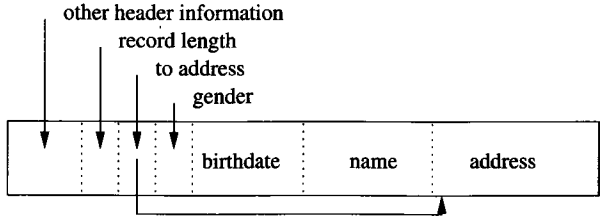

Figure 13.23: A `MovieStar` record with `name` and `address` implemented as variable-length character strings

Representing Null Values

Tuples often have fields that may be NULL. The record format of Fig. 13.23 offers a convenient way to represent NULL values. If a field such as address is null, then we put a null pointer in the place where the pointer to an address goes. Then, we need no space for an address, except the place for the pointer. This arrangement can save space on average, even if address is a fixed-length field but frequently has the value NULL.

13.7.2 Records With Repeating Fields

A similar situation occurs if a record contains a variable number of occurrences of a field F, but the field itself is of fixed length. It is sufficient to group all occurrences of field F together and put in the record header a pointer to the first. We can locate all the occurrences of the field F as follows. Let the number of bytes devoted to one instance of field F be L. We then add to the offset for the field F all integer multiples of L, starting at 0, then L, $2L$, $3L$, and so on. Eventually, we reach the offset of the field following F or the end of the record, whereupon we stop.

Example 13.19: Suppose we redesign our movie-star records to hold only the name and address (which are variable-length strings) and pointers to all the movies of the star. Figure 13.24 shows how this type of record could be represented. The header contains pointers to the beginning of the address field (we assume the name field always begins right after the header) and to the first of the movie pointers. The length of the record tells us how many movie pointers there are. □

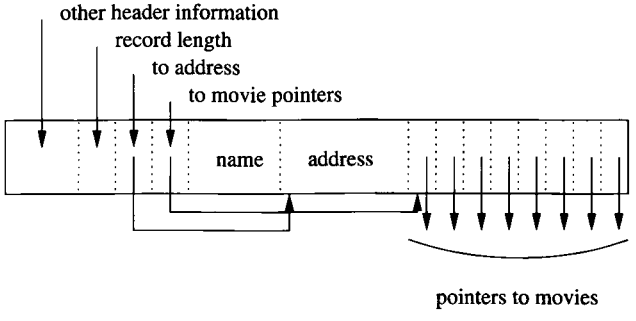

Figure 13.24: A record with a repeating group of references to movies

An alternative representation is to keep the record of fixed length, and put the variable-length portion — be it fields of variable length or fields that repeat

an indefinite number of times — on a separate block. In the record itself we keep:

1. Pointers to the place where each repeating field begins, and
2. Either how many repetitions there are, or where the repetitions end.

Figure 13.25 shows the layout of a record for the problem of Example 13.19, but with the variable-length fields `name` and `address`, and the repeating field `starredIn` (a set of movie references) kept on a separate block or blocks.

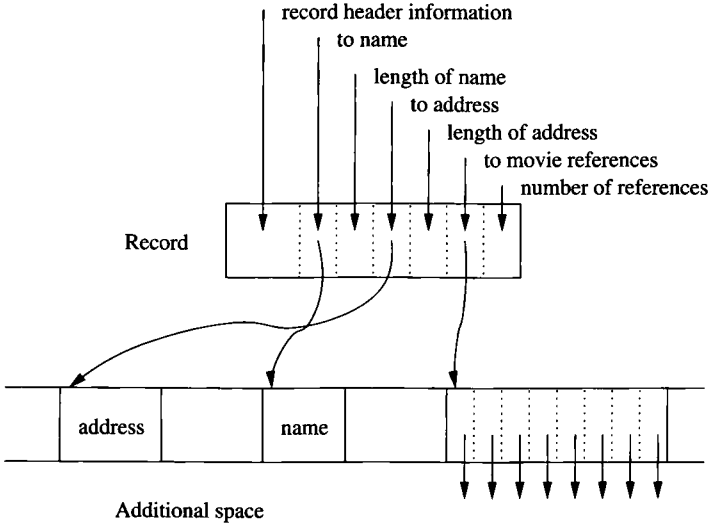

Figure 13.25: Storing variable-length fields separately from the record

There are advantages and disadvantages to using indirection for the variable-length components of a record:

- Keeping the record itself fixed-length allows records to be searched more efficiently, minimizes the overhead in block headers, and allows records to be moved within or among blocks with minimum effort.
- On the other hand, storing variable-length components on another block increases the number of disk I/O's needed to examine all components of a record.

A compromise strategy is to keep in the fixed-length portion of the record enough space for:

1. Some reasonable number of occurrences of the repeating fields,

2. A pointer to a place where additional occurrences could be found, and

3. A count of how many additional occurrences there are.

If there are fewer than this number, some of the space would be unused. If there are more than can fit in the fixed-length portion, then the pointer to additional space will be nonnull, and we can find the additional occurrences by following this pointer.

13.7.3 Variable-Format Records

An even more complex situation occurs when records do not have a fixed schema. We mentioned an example: records that represent XML elements. For another example, medical records may contain information about many tests, but there are thousands of possible tests, and each patient has results for relatively few of them. If the outcome of each test is an attribute, we would prefer that the record for each tuple hold only the attributes for which the outcome is nonnull.

The simplest representation of variable-format records is a sequence of *tagged fields*, each of which consists of the value of the field preceded by information about the role of this field, such as:

1. The attribute or field name,

2. The type of the field, if it is not apparent from the field name and some readily available schema information, and

3. The length of the field, if it is not apparent from the type.

Example 13.20: Suppose movie stars may have additional attributes such as movies directed, former spouses, restaurants owned, and a number of other known but unusual pieces of information. In Fig. 13.26 we see the beginning of a hypothetical movie-star record using tagged fields. We suppose that single-byte codes are used for the various possible field names and types. Appropriate codes are indicated on the figure, along with lengths for the two fields shown, both of which happen to be of type string. □

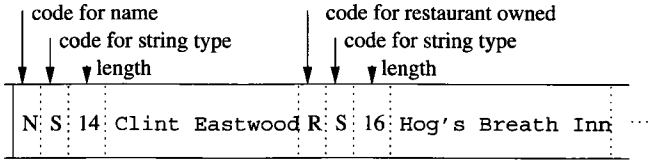

Figure 13.26: A record with tagged fields

13.7.4 Records That Do Not Fit in a Block

Today, DBMS's frequently are used to manage datatypes with large values; often values do not fit in one block. Typical examples are video or audio "clips." Often, these large values have a variable length, but even if the length is fixed for all values of the type, we need special techniques to represent values that are larger than blocks. In this section we shall consider a technique called "spanned records." The management of extremely large values (megabytes or gigabytes) is addressed in Section 13.7.5.

Spanned records also are useful in situations where records are smaller than blocks, but packing whole records into blocks wastes significant amounts of space. For instance, the wasted space in Example 13.16 was only 7%, but if records are just slightly larger than half a block, the wasted space can approach 50%. The reason is that then we can pack only one record per block.

The portion of a record that appears in one block is called a *record fragment*. A record with two or more fragments is called *spanned*, and records that do not cross a block boundary are *unspanned*.

If records can be spanned, then every record and record fragment requires some extra header information:

1. Each record or fragment header must contain a bit telling whether or not it is a fragment.

2. If it is a fragment, then it needs bits telling whether it is the first or last fragment for its record.

3. If there is a next and/or previous fragment for the same record, then the fragment needs pointers to these other fragments.

Example 13.21: Figure 13.27 suggests how records that were about 60% of a block in size could be stored with three records for every two blocks. The header for record fragment $2a$ contains an indicator that it is a fragment, an indicator that it is the first fragment for its record, and a pointer to next fragment, $2b$. Similarly, the header for $2b$ indicates it is the last fragment for its record and holds a back-pointer to the previous fragment $2a$. □

13.7.5 BLOBs

Now, let us consider the representation of truly large values for records or fields of records. The common examples include images in various formats (e.g., GIF, or JPEG), movies in formats such as MPEG, or signals of all sorts: audio, radar, and so on. Such values are often called *binary, large objects*, or BLOBs. When a field has a BLOB as value, we must rethink at least two issues.

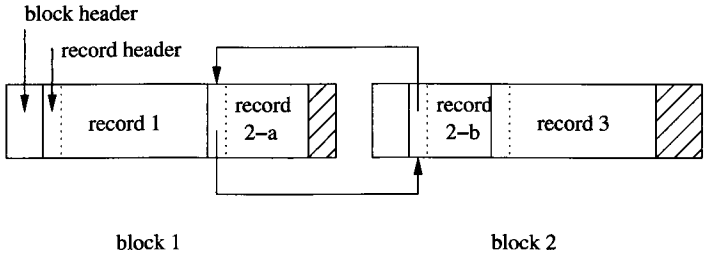

Figure 13.27: Storing spanned records across blocks

Storage of BLOBs

A BLOB must be stored on a sequence of blocks. Often we prefer that these blocks are allocated consecutively on a cylinder or cylinders of the disk, so the BLOB may be retrieved efficiently. However, it is also possible to store the BLOB on a linked list of blocks.

Moreover, it is possible that the BLOB needs to be retrieved so quickly (e.g., a movie that must be played in real time), that storing it on one disk does not allow us to retrieve it fast enough. Then, it is necessary to *stripe* the BLOB across several disks, that is, to alternate blocks of the BLOB among these disks. Thus, several blocks of the BLOB can be retrieved simultaneously, increasing the retrieval rate by a factor approximately equal to the number of disks involved in the striping.

Retrieval of BLOBs

Our assumption that when a client wants a record, the block containing the record is passed from the database server to the client in its entirety may not hold. We may want to pass only the "small" fields of the record, and allow the client to request blocks of the BLOB one at a time, independently of the rest of the record. For instance, if the BLOB is a 2-hour movie, and the client requests that the movie be played, the BLOB could be shipped several blocks at a time to the client, at just the rate necessary to play the movie.

In many applications, it is also important that the client be able to request interior portions of the BLOB without having to receive the entire BLOB. Examples would be a request to see the 45th minute of a movie, or the ending of an audio clip. If the DBMS is to support such operations, then it requires a suitable index structure, e.g., an index by seconds on a movie BLOB.

13.7.6 Column Stores

An alternative to storing tuples as records is to store each column as a record. Since an entire column of a relation may occupy far more than a single block, these records may span many blocks, much as long files do. If we keep the

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 $((1, a), (2, c), (3, e))$ and $((1, b), (2, d), (3, f))$, 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 10 bytes long. It also has the following variable-length fields: name, address, and patient history. If pointers within a record require 4 bytes, and the record length is a 4-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 10–50 bytes; for address it is 20–80 bytes, and for history it is 0–1000 bytes. What is the average length of a patient record?

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 16 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 40 bytes. Also, suppose that for each patient and each test a result is stored with probability p.

a) Assuming pointers and integers each require 4 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 10,000 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:** 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 16 bytes for the fragment header. For what values of r can we improve space utilization by spanning records?

!! **Exercise 13.7.6:** 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.

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.

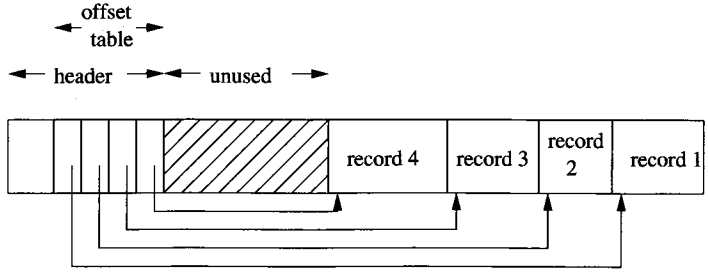

Figure 13.28: An offset table lets us slide records within a block to make room for new records

If we can find room for the inserted record in the block at hand, then we simply slide the records within the block and adjust the pointers in the offset table. The new record is inserted into the block, and a new pointer to the record is added to the offset table for the block. However, there may be no room in the block for the new record, in which case we have to find room outside the block. There are two major approaches to solving this problem, as well as combinations of these approaches.

1. *Find space on a "nearby" block.* For example, if block B_1 has no available space for a record that needs to be inserted in sorted order into that block, then look at the following block B_2 in the sorted order of the blocks. If there is room in B_2, move the highest record(s) of B_1 to B_2, leave forwarding addresses (recall Section 13.6.2) and slide the records around on both blocks.

2. *Create an overflow block.* In this scheme, each block B has in its header a place for a pointer to an *overflow* block where additional records that theoretically belong in B can be placed. The overflow block for B can point to a second overflow block, and so on. Figure 13.29 suggests the structure. We show the pointer for overflow blocks as a nub on the block, although it is in fact part of the block header.

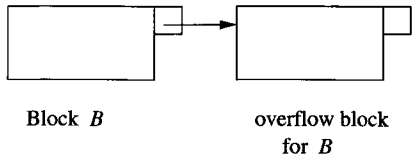

Figure 13.29: A block and its first overflow block

13.8.2 Deletion

When we delete a record, we may be able to reclaim its space. If we use an offset table as in Fig. 13.28 and records can slide around the block, then we can compact the space in the block so there is always one unused region in the center, as suggested by that figure.

If we cannot slide records, we should maintain an available-space list in the block header. Then we shall know where, and how large, the available regions are, when a new record is inserted into the block. Note that the block header normally does not need to hold the entire available space list. It is sufficient to put the list head in the block header, and use the available regions themselves to hold the links in the list, much as we did in Fig. 13.22.

There is one additional complication involved in deletion, which we must remember regardless of what scheme we use for reorganizing blocks. There may be pointers to the deleted record, and if so, we don't want these pointers to dangle or wind up pointing to a new record that is put in the place of the deleted record. The usual technique, which we pointed out in Section 13.6.2, is to place a *tombstone* in place of the record. This tombstone is permanent; it must exist until the entire database is reconstructed.

Where the tombstone is placed depends on the nature of record pointers. If pointers go to fixed locations from which the location of the record is found, then we put the tombstone in that fixed location. Here are two examples:

1. We suggested in Section 13.6.2 that if the offset-table scheme of Fig. 13.28 were used, then the tombstone could be a null pointer in the offset table, since pointers to the record were really pointers to the offset table entries.

2. If we are using a map table, as in Fig. 13.18, to translate logical record addresses to physical addresses, then the tombstone can be a null pointer in place of the physical address.

If we need to replace records by tombstones, we should place the bit that serves as a tombstone at the very beginning of the record. Then, only this bit must remain where the record used to begin, and subsequent bytes can be reused for another record, as suggested by Fig. 13.30.

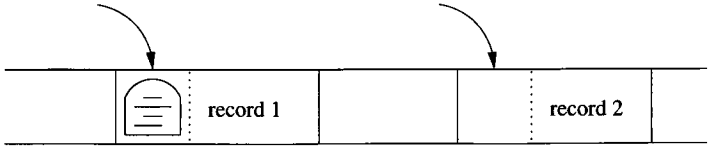

Figure 13.30: Record 1 can be replaced, but the tombstone remains; record 2 has no tombstone and can be seen when we follow a pointer to it

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 three 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

the proper cylinder, the rotational latency during which the desired block rotates under the head, and the transfer time, while the block moves under the head and is read or written.

- *Speeding Up Disk Access*: There are several techniques for accessing disk blocks faster for some applications. They include dividing the data among several disks (striping), mirroring disks (maintaining several copies of the data, also to allow parallel access), and organizing data that will be accessed together by tracks or cylinders.

- *Elevator Algorithm*: We can also speed accesses by queueing access requests and handling them in an order that allows the heads to make one sweep across the disk. The heads stop to handle a request each time it reaches a cylinder containing one or more blocks with pending access requests.

- *Disk Failure Modes*: To avoid loss of data, systems must be able to handle errors. The principal types of disk failure are intermittent (a read or write error that will not reoccur if repeated), permanent (data on the disk is corrupted and cannot be properly read), and the disk crash, where the entire disk becomes unreadable.

- *Checksums*: By adding a parity check (extra bit to make the number of 1's in a bit string even), intermittent failures and permanent failures can be detected, although not corrected.

- *Stable Storage*: By making two copies of all data and being careful about the order in which those copies are written, a single disk can be used to protect against almost all permanent failures of a single sector.

- *RAID*: These schemes allow data to survive a disk crash. RAID level 4 adds a disk whose contents are a parity check on corresponding bits of all other disks, level 5 varies the disk holding the parity bit to avoid making the parity disk a writing bottleneck. Level 6 involves the use of error-correcting codes and may allow survival after several simultaneous disk crashes.

- *Records*: Records are composed of several fields plus a record header. The header contains information about the record, possibly including such matters as a timestamp, schema information, and a record length. If the record has varying-length fields, the header may also help locate those fields.

- *Blocks*: Records are generally stored within blocks. A block header, with information about that block, consumes some of the space in the block, with the remainder occupied by one or more records. To support insertions, deletions and modifications of records, we can put in the block header an offset table that has pointers to each of the records in the block.

- *Spanned Records*: Generally, a record exists within one block. However, if records are longer than blocks, or we wish to make use of leftover space within blocks, then we can break records into two or more fragments, one on each block. A fragment header is then needed to link the fragments of a record.

- *BLOBs*: Very large values, such as images and videos, are called BLOBs (binary, large objects). These values must be stored across many blocks and may require specialized storage techniques such as reserving a cylinder or striping the blocks of the BLOB.

- *Database Addresses*: Data managed by a DBMS is found among several storage devices, typically disks. To locate blocks and records in this storage system, we can use physical addresses, which are a description of the device number, cylinder, track, sector(s), and possibly byte within a sector. We can also use logical addresses, which are arbitrary character strings that are translated into physical addresses by a map table.

- *Pointer Swizzling*: When disk blocks are brought to main memory, the database addresses need to be translated to memory addresses, if pointers are to be followed. The translation is called swizzling, and can either be done automatically, when blocks are brought to memory, or on-demand, when a pointer is first followed.

- *Tombstones*: When a record is deleted, pointers to it will dangle. A tombstone in place of (part of) the deleted record warns the system that the record is no longer there.

- *Pinned Blocks*: For various reasons, including the fact that a block may contain swizzled pointers, it may be unacceptable to copy a block from memory back to its place on disk. Such a block is said to be pinned. If the pinning is due to swizzled pointers, then they must be unswizzled before returning the block to disk.

13.10 References for Chapter 13

The RAID idea can be traced back to [8] on disk striping. The name and error-correcting capability is from [7]. The model of disk failures in Section 13.4 appears in unpublished work of Lampson and Sturgis [5].

There are several useful surveys of disk-related material. A study of RAID systems is in [2]. [10] surveys algorithms suitable for the secondary storage model (block model) of computation. [3] is an important study of how one optimizes a system involving processor, memory, and disk, to perform specific tasks.

References [4] and [11] have more information on record and block structures. [9] discusses column stores as an alternative to the conventional record

structures. Tombstones as a technique for dealing with deletion is from [6]. [1] covers data representation issues, such as addresses and swizzling in the context of object-oriented DBMS's.

1. R. G. G. Cattell, *Object Data Management*, Addison-Wesley, Reading MA, 1994.

2. P. M. Chen et al., "RAID: high-performance, reliable secondary storage," *Computing Surveys* **26**:2 (1994), pp. 145–186.

3. J. N. Gray and F. Putzolo, "The five minute rule for trading memory for disk accesses and the 10 byte rule for trading memory for CPU time," *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pp. 395–398, 1987.

4. D. E. Knuth, *The Art of Computer Programming, Vol. I, Fundamental Algorithms, Third Edition*, Addison-Wesley, Reading MA, 1997.

5. B. Lampson and H. Sturgis, "Crash recovery in a distributed data storage system," Technical report, Xerox Palo Alto Research Center, 1976.

6. D. Lomet, "Scheme for invalidating free references," *IBM J. Research and Development* **19**:1 (1975), pp. 26–35.

7. D. A. Patterson, G. A. Gibson, and R. H. Katz, "A case for redundant arrays of inexpensive disks," *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pp. 109–116, 1988.

8. K. Salem and H. Garcia-Molina, "Disk striping," *Proc. Second Intl. Conf. on Data Engineering*, pp. 336–342, 1986.

9. M. Stonebraker et al., "C-Store: a column-oriented DBMS," *Proc. Thirty-first Intl. Conf. on Very Large Database Systems*" (2005).

10. J. S. Vitter, "External memory algorithms," *Proc. Seventeenth Annual ACM Symposium on Principles of Database Systems*, pp. 119–128, 1998.

11. G. Wiederhold, *File Organization for Database Design*, McGraw-Hill, New York, 1987.