The disk layout problem

John DeTreville
Microsoft Corporation
johndetr@microsoft.com
+1 425 936-0544

Abstract

A disk drive contains a large number of fixed-sized blocks that can hold data. We can choose to map a given collection of data records to these blocks in many ways, but different data layouts will result in different disk performance. This paper presents a series of increasingly accurate models of disk drives and their performance, and poses the open question of how to find "good" layouts in practice.

Introduction

Computers use both RAM[1] and disks[2] to store data. RAM operates completely electronically while disks are largely mechanical in nature. RAM is much faster than disks, but disks can typically hold much more data.[3] The cost of accessing data in RAM is largely independent of its layout in RAM,[4] while the cost of accessing data on a disk depends quite strongly on its layout.

Let's be concrete. A typical new desktop computer in late 2001 might have 128MB of RAM and 40GB of disk.[5] Each disk block holds 512 bytes, so the disk contains about 80 million blocks. (We note empirically that disks on desktop computers are about 50% full; there are about half as many data blocks as disk blocks.) The processor can read a 512-byte data record from RAM in about 4 microseconds, while reading a data record from disk might have a latency of between 2 millisecond and 20 milliseconds, depending on the layout of the data records on disk; the slowdown is between $500\times$ and $5{,}000\times$. Choosing a good layout can therefore, in principle, speed up disk reads by up to $10\times$ over a bad layout.[6] This is important because disks are often the performance bottleneck for the users of desktop computers.

A simple disk model

One very simple way to model a disk is as a one-dimensional array of fixed-sized blocks, as shown in Figure 1. The blocks are numbered B_0, B_1, B_2, …, and the data records to be stored are numbered D_0, D_1, D_2, …. Each block holds zero or one data records; each data

[1] "Random-Access Memory," usually implemented using integrated circuitry.
[2] Disks are also called "disk drives" or "hard drives."
[3] RAM is also volatile, meaning that its contents disappear when powered off, while disks are nonvolatile. All data that is intended to be persistent must be stored on disk.
[4] This is true only for a uniform store of RAM. Most computers have multiple levels of RAM-like memory, with smaller, faster "caches" complementing the "main memory." Allowing the proper distribution of data among these multiple levels is itself a largely open problem.
[5] A megabyte is a million (10^6 or 2^{20}) bytes, and a gigabyte is a billion (10^9 or 2^{30}) bytes. In this paper, a byte is a collection of 8 bits.
[6] In fact, the speedup can be much greater due to the use of track buffers, as described later in this paper.

record must reside in at least one disk block. (The number of data records does not exceed the number of disk blocks.) The disk has a single *head* that may be moved among the blocks;[7] here the head is shown positioned at block B_2.

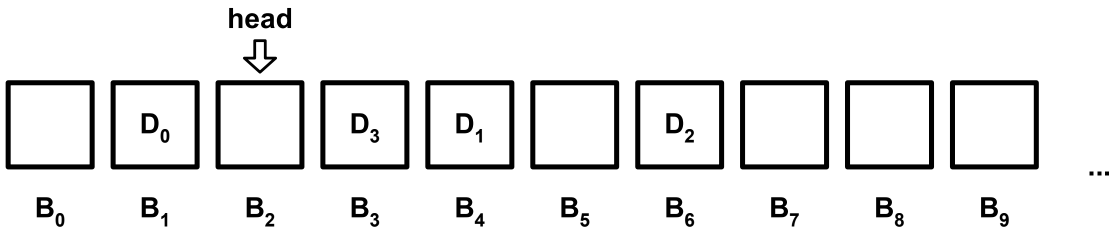

Figure 1. A one-dimensional array of fixed-sized blocks.

A *trace* is a sequence of references (reads or writes) to data records or to disk blocks. For example, D_0 D_1 D_2 D_3 is a data trace, corresponding here uniquely to the disk trace B_1 B_4 B_6 B_3.

Executing a trace (or executing the computer software that produces the trace) requires moving the head to each of the disk blocks named, in turn. For initial simplicity, let us assume that the cost of accessing a block equals the minimum number of unit steps that the head must travel.[8] The cost of executing this trace is therefore 9 steps for the data layout and initial head position shown above.

For a given trace or a stochastic process that generates possible traces, the best layouts are those that minimize the (expected) time required to execute a trace, or perhaps those that also reduce the variance in the presence of uncertainty. Finding the best layouts may be intractable, and so we are also interested in the problem of finding a "good" layout, or at least avoiding "bad" layouts. Although many heuristics have been developed over the years to avoid particularly bad layouts, these have no particular foundation in theory, and so new layout strategies with a strong theoretical foundation are of particular interest.

Simplifications

If we use the simple model above, a few simple results are obvious.

We note that data records should always be packed contiguously on the disk; there should be no holes since holes never improve the performance of a trace.[9] If no data record is to appear on disk more than once, the layout problem then reduces to finding an optimal permutation of the data records.[10]

A further simplification is possible if we note that data records usually constitute *files,* with each file containing a sequence of data records. Each data record is contained in at

[7] Moving the head is also called "seeking" the head.
[8] If this were in fact the case, the ratio between the worst-case and best-case read times would clearly exceed the 10:1 ratio described above for real disks.
[9] Of course, the presence of holes may help in the future allocation of new disk blocks that are close to existing blocks.
[10] While replicating data records on disk can speed up reading, it can slow down writing those records. More disk blocks must be updated, and additional information must be stored on disk during these updates so that the disk can be restored to a consistent state following an ill-timed system failure. The right replication policy therefore depends on the ratio of reads to writes.

most one file, and appears in that file only once. If files are always accessed sequentially and in their entirety, with no intervening accesses to other data records, then files should clearly be laid out contiguously on disk. Programs called *disk defragmenters* rearrange data records on disk so that each file is contiguous, with no holes or few holes between data records.[11] Some disk defragmenters also try to place related files near each other, usually based on simple static structure rather than a dynamic analysis of the accesses.[12]

If there is only one trace to optimize for, and the length of the trace does not exceed the number of blocks on the disk, then the layout of data records on the disk should follow the order in the trace. For the trace D_0 D_1 D_2 D_3, the optimal data layout is as shown in Figure 2; the cost of executing the trace is now just 3 steps.

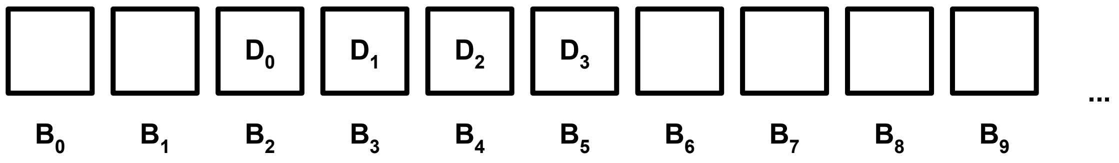

Figure 2. An optimal data layout for one trace.

This simple result, of course, does not apply if more than one trace is possible. For example, if the traces D_0 D_1 D_2 D_3 and D_3 D_1 D_4 D_1 D_5 D_9 are equally likely, and the initial head position is still at block B_2, an optimal disk layout is shown in Figure 3.[13] The expected execution time is now 6½ steps.

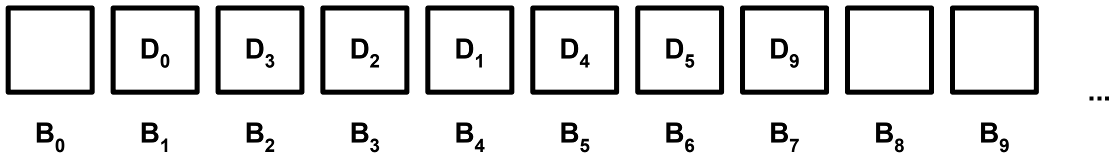

Figure 3. An optimal data layout for two equally likely traces.

Even with this simple disk model and performance model, finding an optimal layout might be intractable in the general case. It would therefore be useful to understand how to find a good layout for a given set of data records and a given class of traces. It would also be useful to understand the limits of such an analysis.

Complexities

The disk layout problem as stated is perhaps complex enough, but we list here a number of further complexities, some of which might enrich possible solutions.

[11] An alternative to using a disk defragmenter is to adopt an online strategy for file placement that avoids fragmentation. This is difficult because files often grow after they are first allocated, and the additional disk records cannot always follow the original records.

[12] Files are typically organized into collections denoted by *directories;* directories can also include other directories. Like files, directories are represented by a number of data records held in disk blocks. File allocators and disk defragmenters sometimes try to keep directories and the files they contain close to each other on disk because of access patterns that are known to be typical.

[13] This particular example disk layout was discovered using exhaustive search, which is not practical in general.

Observing traces

Although we spoke above of a stochastic process generating possible traces, we generally do not have direct knowledge of that process to work with. Instead, we can merely observe some number of actual traces, and possibly reason further about common future access patterns based on our knowledge of the software and its users.[14]

One approach might be to use a number of observed traces to build a simple model of common traces or subtraces, and then to use that model to find good layouts. An alternative might be to use the observed traces directly to find good layouts.

One added complexity is that the computer may be running several independent programs concurrently, and the different traces for different programs may be nondeterministically interspersed. The layout algorithm should not be overly sensitive to such possible nondeterminism.

Multiple outstanding requests

Our model so far has been that disk accesses must be performed according to some total ordering. We might relax this to a partial ordering. For example, we might say that at any moment there can be multiple disk accesses outstanding, which may be executed in any convenient order. If multiple independent programs on the computer wish to access the disk, the order in which these accesses are executed might not be important, and some orders might perform better than others. Similarly, if we wish to read a file in its entirety, the order in which its data records are read might not matter.

A known good dynamic heuristic, for a given disk layout, is to reorder outstanding access requests so that the disk head seldom changes its direction of travel. It might be possible to choose a disk layout that interacts especially well with this heuristic.

Read-ahead

At any point in time, it can be useful to guess what future disk reads may occur, and perform the reads before they are requested. For example, if we read the first data record of a file, we might expect that the second record will soon be read. Reading it now can obviously make sense if the disk is otherwise idle, or if the incremental cost of doing so is very small.

Again, it may be possible to choose a disk layout that interacts especially well with dynamic read-ahead. Moreover, the same predictive information that is used for disk layout might be used to direct read-ahead.

Caching data records in RAM

If the same data records are frequently read from disk, it can be advantageous to keep copies of these records in RAM. One implementation is to retain the k most recently used

[14] For example, we might know that a program has a lengthy initialization phase followed by an interactive phase driven by user inputs. If the initialization phase is completely or largely independent of user input, it might make sense to optimize for it separately. In fact, a reasonable heuristic might be to optimize disk layout solely on the initialization trace, since the user might be much less sensitive to disk performance during the interactive phase following.

data records, avoiding the need to reread them. There may be disk layouts that interact particularly well with such a dynamic caching policy.

Online layout

A data layout algorithm might operate in a batch mode, working with a number of traces that have already been collected, or it might operate in an online mode, where it makes incremental adjustments to the disk layout based on limited memory of recent accesses. One simple strategy might be, whenever moving the head from B_i to B_j (assume $i < j - 1$), to swap the contents of B_j and B_{j-1}, thereby moving the requested data records closer together.

This example algorithm has several obvious bad properties,[15] but other online algorithms may be able to improve disk layout continuously. This is especially important if access patterns change over time, since online algorithms could immediately begin to make improvements.

3-D geometry

While we have modeled disks as one-dimensional arrays of blocks, the truth is that real disks have a 3-D geometry.

A disk contains a small number of 2-D platters (typically 1–5) stacked on a central spindle. Each platter is a disk with blocks on both sides (“surfaces”) and holds about 20GB. The disks rotate together at high speed, typically completing a rotation in about 8 milliseconds.

Each surface has tens of thousands of concentric circular tracks; a given distance from the center defines a “cylinder” of tracks from among all the surfaces. An assembly of heads, one per surface, moves in unison along a radius.[16] The heads can be moved to any cylinder. A head movement takes up to 15 milliseconds.[17]

(It is also increasingly common for multiple disks to be ganged together in configurations where data records can be laid out across the disks. This effectively increases the number of independent disk head assemblies, and can lead to improved performance or reliability. Such configurations are however not yet common in desktop computers.)

Each track is further subdivided into blocks.[18] Once the head assembly is positioned, there is a brief rotational latency until the desired block rotates past the head, at which time the block is accessed. The access time for a block includes the variable head-movement latency, plus a variable rotational latency (0–8 milliseconds), plus smaller fixed costs.

[15] For example, it adds two reads and two writes to every disk access, even without counting the extra writes required to recover in case of a system failure after the first write but before the second. Moreover, it is not clear that this algorithm converges to an especially good layout.

[16] Actually, along a curved path close to a radius.

[17] The head assembly accelerates and decelerates at a constant acceleration, up to a maximum velocity. There is also a fixed period following a head movement for the heads to settle into their final position. One head is active at once and switching heads also involves a small settling period.

[18] These blocks are also called sectors.

Accessing sequential blocks on the same track was clumsy on early disks, since the second block would often have rotated past the head before the software could request it. Modern disks contain electronic track buffers that can hold a large number of blocks, and the blocks that pass the heads following a read are automatically cached into the track buffer to satisfy future read requests.

Because the circumference of the tracks varies, there are more disk blocks on the outside tracks than on the inside tracks; the ratio is approximately 2:1. Disk accesses are faster on the outer tracks because more blocks pass the heads every second,[19] and because the greater capacity makes it less often necessary to move the heads.

As a final complication, the actual disk geometry is usually not exposed to the computer, which sees only a simplified geometry. This is for reasons of backward compatibility, and also allows the disk to automatically remap faulty disk blocks without the involvement of the software. The outlines of the actual disk geometry can be determined experimentally but the details possibly cannot.

Summary

The layout of data on disk drives can greatly affect the performance of data access. A simple model of disk performance leads to a simple combinatorial view of data layout, although our model becomes more complex as we model the performance and geometry of real disk drives with increasing accuracy.

Because finding an optimal layout may be intractable, and because access patterns change over time, we would particularly like to find simple adaptive algorithms that produce relatively good layouts.

[19] When the heads move among the cylinders, the angular velocities of the platters do not change. All sectors have roughly the same density of bits per linear inch.