Chapter 1

The Disk Layout Problem

Brian Corbett[1], Gregory Dresden[2], Nancy Ann Neudauer[3], Marc Paulhus[4],
Report prepared by Nancy Ann Neudauer

We can organize data on a personal computer's hard drive according to many different data strategies resulting in different performances due to disk latencies, consisting of both rotational latency and seek time. Rotational latency is a physical characteristic of the disk and motor, so we focus on the problem of storing data in a manner that optimizes the seek time of the data. The optimization of this problem will result in better performance for users.

1.1 Introduction

Imagine that we keep a daily log of the files that our computer reads from its hard disk. For most computer users the logs of one day compared to the next may be very similar. For example, opening up a commonly used program may require access to the same files in the same order every time that event occurs. We shall call such a sequence of files a *trace*. Our daily log is composed of a large number of different traces. However, there is good reason to belive that some traces will appear largely unchanged in our log from day-to-day and perhaps multiple times in a single daily log.

Now imagine that our disk is a random ad-hoc jumble of files in no particular order (this should not be too hard for most of us to imagine). Our computer, performing the tasks we ask of it, may have to work very hard to access the files in the order that they are required. If files that are adjacent in a common trace are stored far apart on the disk then we should expect that our disk performance will be poor. On the other hand, if we rearranged our disk in such a way that those files were close together, we should expect improved performance.

This is the essence of the problem Microsoft posed to the PIMS 5th Industrial Problem Solving Workshop. Given a set of traces that are expected to be representative of common use, we must rearrange the files on the disk so that the performance is optimized.

[1]University of Manitoba
[2]Washington and Lee University
[3]Pacific University
[4]Pacific Institute for the Mathematical Sciences

Some immediate observations are clear. One is that all the parts of a single file should be contiguous (assuming that the computer only has uses for complete files). A second is that it can not help our disk performance to have gaps of data on our disk; gaps can only increase the distances between files.

Programs called *disk defragmenters* use these simple principles to rearrange data records on a disk so that each file is contiguous, with no holes or few holes between data records. Some more sophisticated 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. We are interested in more dynamic defragmentation procedures.

We first consider a 1D model of the disk. We then look at the results from an investigation of the 2D disk model followed by a discussion of caching strategies. Finally we list some of the complications that may need to be addressed in order to make the models more realistic.

1.2 1D Disk Layout Model

One way to model the disk is to imagine it as having only a single (circular) track, with blocks on that track labeled $B_0, B_1, \ldots, B_n$ where block B_n is followed by block B_0, then block B_1, and so on, creating a cycle. The *files*, say $D_0, D_1, \ldots D_k$, are placed inside these blocks. The *head* sits in a fixed location and the disk spins (in one direction, for our purposes counterclockwise). The head can read the file that is directly beneath it. See Figure 1.1.

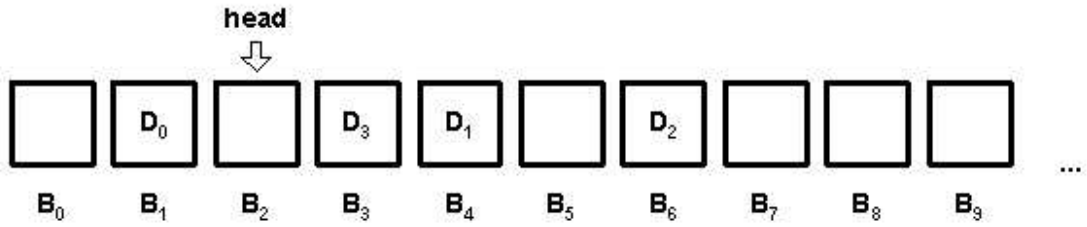

Figure 1.1: A one-dimensional array of fixed-sized blocks.

Our task is to rearrange the files that are assigned to each block to minimize the *cost* on a given trace. The *cost* is simply a count of the number of blocks which must pass under the head while it is reading the given trace.

Suppose that the trace in question is $\{D_0\ D_1\ D_2\ D_3\}$. Then from the starting position shown in Figure 1.1, the cost is 21. Is there a better layout that would reduce the cost?

In this case, of course, the solution is obvious. Since there is only one trace and each file appears exactly once, the optimal data layout is the trace itself, as shown in Figure 1.2. The cost of executing the trace is now just 3.

When there is more than one trace or when the same file appears multiple times in the same trace then the situation gets more complicated. As a model for this scenario, consider a complete directed graph where the nodes are the files and each directed edge (F_1, F_2) is assigned a cost function based on the number of times that file F_1 is followed by file F_2 in the given trace.

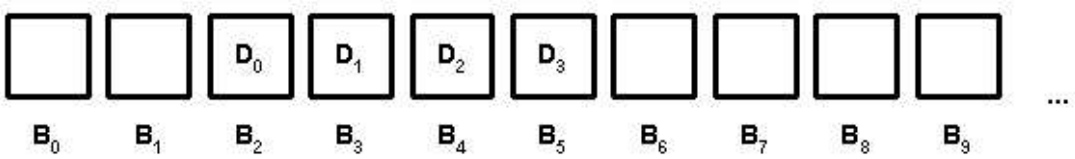

Figure 1.2: An optimal data layout for $\{D_0\ D_1\ D_2\ D_3\}$ trace

A good layout to consider is the maximal tour on this graph. Thus we can see that the problem is closely related to the famous NP-hard traveling salesman problem (see, for example, Cormen, Leiserson and Rivest, *Introduction to Algorithms*, MIT Press (1995), pp. 969).

This 1D disk model is not a particularly accurate representation as most disks consist of a number of concentric tracks, sometimes on both sides of the disk, and sometimes with a platter of disks stacked one atop the other. However, it is an unfortunate reality that most disks in use today do not reveal their precise geometry to the operating system. Instead, they reveal a 1D geometry not unlike our simple linear model. In this common case the 1D model is the only option available.

We now describe some heuristic methods we use to investigate the problem.

The following assumptions are made:

1. All files are the same size and will fit exactly in one block.

2. The disk is completely packed. That is, there are no empty blocks (this is relaxed slightly by necessity in Section 1.3).

3. The disk spins at a constant rate.

4. There is a cache of size one. That is, if file F_1 has just been read and the trace asks to read F_1 again, then there is no cost for this. More on caching will be discussed in Section 1.4.

5. Every file appears on the disk exactly once. It may seem tempting to duplicate commonly used files to improve disk performance. However, if overused this technique will quickly fill a disk. Also, the time required to update or change a file will increase.

1.2.1 1D Disk Layout Results

Consider a set of fixed traces, each consisting of a certain number of files. We seek a new arrangement of these blocks such that the cost function, applied to each trace, is reasonably low.

If we had world enough and time (to quote Andrew Marvell) we could look at every possible permutation of blocks, calculate the cost of each trace on each permutation, and thus find the best arrangement. This is obviously impractical, so we need to come up with a faster way to calculate the cost, and a better way to find a good arrangement of blocks.

Let us first define and discuss the adjacency matrix, which gives us a quick way of judging the worthiness of a particular configuration of the blocks. We define A to be an $n \times n$ matrix, initially all entries 0, and indexed by the blocks in the trace. For each consecutive pair of blocks i, j in the traces, we increment the corresponding matrix entry $A_{i,j}$ by 1. So, given the trace $T = \{d, c, b, a, d, b, c, a, a, d, c, a, d, d, d, b, a, c, b, a\}$, and with rows and columns labelled in the order a, b, c, d, the matrix A is

	a	b	c	d
a	.		1	3
b	3	.	1	
c	2	2	.	
d		2	2	.

We replace the diagonal entries with "." both for ease of reading and to illustrate that there is no cost associated with accessing the same block twice in a row.

Clearly, the initial block configuration of a, b, c, d for the trace T (with a cost of 41) is far from optimal here: we see from our matrix that a is never followed by b, nor is c followed by d (as $A_{a,b} = A_{c,d} = 0$). However, the pair b, a occurs three times, as $A_{b,a} = 3$. We seek a block configuration that gives an adjacency matrix with large numbers on the upper diagonal and small numbers on the lower diagonal, thus indicating that commonly-occuring (respectively rarely-occuring) pairs of blocks in the trace T will actually be adjacent (respectively, far apart) in the new block configuration. In this case, a better configuration might be d, c, b, a with adjacency matrix:

	a	b	c	d
a	.	3	2	
b		.	2	2
c	1	1	.	2
d	3			.

The cost is easily calculated to be 23, a nice improvement.

We notice that one advantage of the adjacency matrix is that it allows us to quickly calculate the cost of a particular configuration of blocks. The explicit formula is

$$\text{cost} \ = \sum_{i=0}^{n-1} \sum_{j=1}^{n} i \cdot A_{j,1+(i+j-1 \mod n)}$$

Now that we can measure the effectiveness of a particular permutation of blocks, let us discuss how to find a configuration that reduces the cost. First, we employ a greedy algorithm that searches for the pair of blocks that occur together most often (say, x and y) and places them together in locations 1 and 2. Then, we find which block follows y most often, and we place it in block 3, and so on. This is an extremely fast and efficient method, and in practice this can reduce cost by as much as 25%, depending on the initial conditions. Second, we use the method of simulated annealing, in which we randomly permute pairs of blocks, re-calculate

the cost, and decide whether or not to keep the new configuration. If the cost is lower, then we keep the new layout; if the cost is higher, we evaluate $e^{-d/t}$, with $d =$ the difference in cost and $t =$ the current *temperature,* a value which initially is quite large but decreases at each step. If the $e^{-d/t}$ is greater than a random number between 0 and 1, we admit the new, higher-cost configuration, but if not, we retain the original layout. Early in the algorithm, the temperature t is set to be quite *hot,* and so a fair amount of randomness is tolerated; as the temperature is lowered and the algorithm *cools down,* the layout settles on a nice configuration of low cost. This process is repeated – the temperature is again raised, then cooled down, and a configuration of low cost is found.

Together, these methods are an efficient way to find a cost-effective ordering of disk blocks that, we hope, will speed up access time for the user. As an illustration, we ran a simulation with $n = 100$ blocks, and five traces of length 500 each. The traces were mostly random, except that in an attempt to simulate a typical log of disk access activity there was a one-in-three chance that a particular number, k, would be followed by $2k + 1 \bmod n$. Thus, the simulation represented about 2500 different visits to the (10000 total) pairs of blocks on the disk, meaning that almost every pair i, j occurs no more than three times (and most pairs happen once or not at all). The cost for the initial disk layout was 121505. Application of the greedy algorithm brought the cost down to 109027, and simulated annealing brought it down further to 90929, for a total savings of about 25%.

Realizing that the above might not be the best model for disk access, we constructed another simulation. Again, we considered $n = 100$ blocks, but this time we randomly selected 200 pairs of blocks, and had each pair appear in our trace (of disk activity) a random number of times, up to 50. Thus, in this simulation we were modelling about 5000 different visits, twice as many as above, but not nearly as broadly dispersed. In this case, our starting cost was 245684, which was brought down to 190414 by the greedy algorithm and then to 103253 by simulated annealing, a savings of almost 60%.

We see that the effectiveness of our procedure depends heavily on the type of data; if the disk activity consists of visiting a large number of disparate blocks, without much repetition, then the procedure outlined above is not particularly good at finding a good configuration. (Indeed, in such a scenario it is hard to imagine how any procedure could do very well.) Fortunately, most disk activity involves repeated visits to the same sequence of blocks, and in this case our algorithm can offer significant savings.

1.3 2D Disk Layout

In reality a collection of stacked disks comprise a hard drive, not a 1D array of blocks. Each disk consists of a series of blocks laid-out on concentric tracks on a circular disk similar to Figure 1.3. As a disk spins, the read-head moves back and forth along a fixed radial line. Note that the number of blocks along the outside of the disk is greater than the number of blocks along the inside of the disk.

For computational simplicity we assume that the number of blocks in a given row (or track) is independent of the distance from the center of the disk. Also, rather than having the disk spin, we take the equivalent view that the head is moving on the disk in a single direction. From

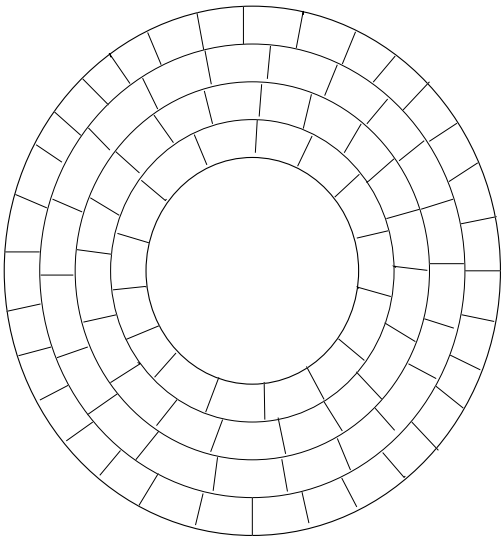

Figure 1.3: A 2D-Layout of blocks on a disk

any given cell the head can move to any adjacent cell in the next column (as in Figure 1.4).

Real disks have about twice as many rows as columns (unlike our diagrams). Finally, we restrict ourselves to considering a single disk rather than a stack of disks.

An immediate observation one can make is that the performance of even a random layout should be greatly improved in the 2D model over the 1D model. Simply put there are more files close together in the 2D model. For example, from a given file in the 2D model there are three files which can be accessed with a cost of 1, whereas in the 1D model there is only one!

A good 1D layout can be transformed into a 2D model simply by "wrapping" the files around the disk, starting in the outside and ending in the inside. See Figure 1.5.

However, given a random 2D layout we can improve on the performance of the disk by applying simulated annealing directly to the 2D geometry. Table 1.1 summarizes our results. The trace we used was extracted from some actual disk logs kindly provided by John DeTreville of Microsoft Research.

We see that our heuristic optimization techniques appear to perform better when applied

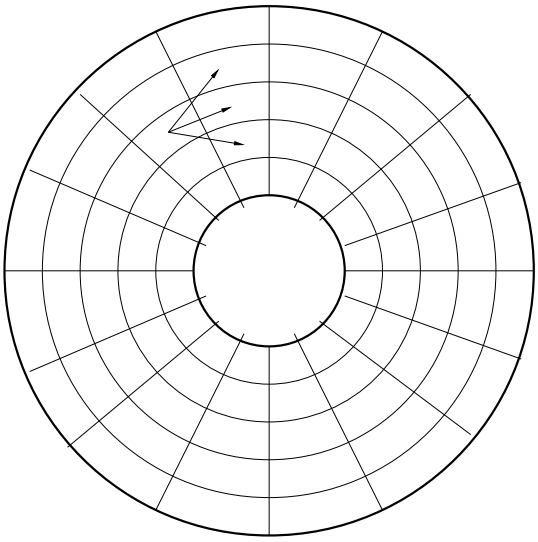

Figure 1.4: Our 2D disk model

directly to the 2D geometry than when applied them to the 1D geometry and then transformed to 2D. This suggests that when performance is critical it is better to optimize the 2D geometry directly. The main problem is, however, that modern hardware only provides access to the 1D geometry of the disk. Our results suggest that disk performance can be improved if 2D (or possibly even 3D) information were available.

1.4 Caching Strategies

If the same data records are frequently read from disk, it can be advantageous to keep copies of these records in RAM. This is called the *cache*. One strategy for deciding which records should be in the cache is to retain the k most recently used data records, avoiding the need to reread them. There may be disk layouts that interact particularly well with such a dynamic caching policy.

Our model for the RAM cache is simple. We assume that the cache consists of k block-sized

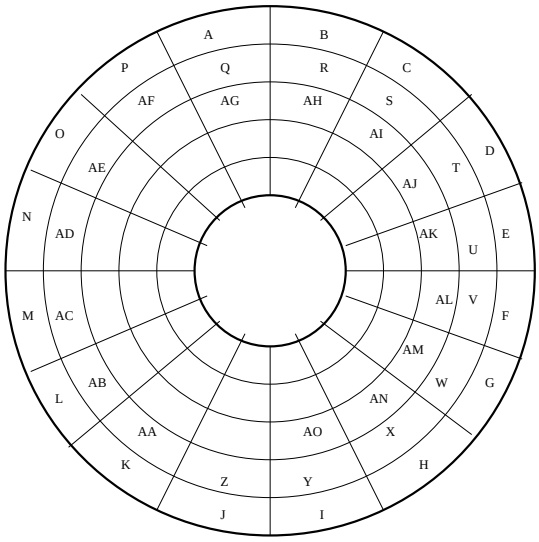

Figure 1.5: The 1D layout $\{A, B, C, ...AM, AN, AO\}$ is transformed into a 2D layout by "wrapping" the files around the disk.

memory chunks in a queue. The difference between cache memory and disk memory is that cache memory is *free* in the sense that it takes no access time to read the files in the cache. Every time a file is requested in our trace, we check to see if it is in the cache queue. If it is, then we consider the file as read and we move on to the next file in the trace. Files read from the disk will be placed in the queue. Of course, since the queue is finite, we have to decide which file to remove from the cache when we add one. There are a number of different strategies for managing the cache. We investigated four:

- A random strategy: a random file in the cache is removed to make room for the new file.
- A FIFO (first-in-first-out) strategy: the file which has been in the queue the longest is deleted to make room for the new file.
- A LRU (least-recently-used) strategy: the file in the queue which was accessed least recently is deleted to make room for the new file (note that this differs from the FIFO

Optimization	Cost
Random Layout	18314
Best 1D Layout transformed	16851
Simulated Annealing on 2D model	10900

Table 1.1: The average performance of a given thread under various optimization strategies.

strategy because a file that is accessed in the queue will be moved to the front). This strategy is the industry standard.

- A conditional strategy: the file in the queue which is least likely to be accessed next is removed to make room for the new file. Note that this strategy involves maintaining a probability-transition-matrix to keep track of which files are most likely to be accessed next. This adds significant overhead to the cache management strategy.

Strategy	Average Cost
Random	165000
FIFO	148500
LRU	144000
Conditional	143000

Table 1.2: Results of different Caching Strategies.

The results of applying these four strategies can be seen in Table 1.2. We applied the strategies to 200 random layouts of the 1D disk models. From the table we can see that the conditional strategy was the best; however it was only marginally better than the industry-standard LRU strategy. Given the additional overhead required to apply the conditional strategy, we conclude the LRU strategy is the best of those we considered.

No attempt was made to optimize the disk layout for given caching strategies. Indeed, in the results reported for 1D and 2D disk models we assume that there is a simple cache of size one, the cached file is always discarded when a new file is read.

1.5 Added layers of complexity to the model

1.5.1 Multiple outstanding requests

In our model so far we have assumed 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.

It can be useful to guess what future disk reads may occur and to 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 to establish the disk layout might be used to direct read-ahead.

1.5.2 Exact 2D Geometries

Our 2D model assumed that each disk track had the same number of blocks. This is not true and the actual geometry of the disk adds a non-trivial complication to the model. Moreover, since the industry standard is not to report the details of the disk geometry to the operating system, only limited optimization may be possible.

1.5.3 Disk and Head Speed

In our model the disk was spinning at a constant rate. Indeed, this is not quite true. Disks stop, speed up, and slow down. The head accelerates and decelerates when it has to scan the surface of the disk. These factors could be substantial.

1.5.4 Similarity of Use

The assumption that traces that appear in one disk log are likely to appear again in future, or even the less strict assumption that current disk use is a good indicator of future disk use, is very strong. Before a great deal of effort is invested into disk layout optimization, some investigation of the validity of these assumptions should be made.

1.6 Acknowledgements

The authors would like to thank the Pacific Institute for Mathematical Sciences for hosting the workshop that led to this paper, and we also express our gratitude to John DeTreville of Microsoft Research for posing the disk layout problem and for providing us with invaluable assistance throughout the workshop.