File design fallacies

S. J. Waters

LSE London WC2

Many systems publications and courses refer to hearsay rules of thumb which supposedly simplify the complex procedure of designing data processing files; consequently, these rules are widely used in practice. In the apparent absence of any published justifications for these rules, this paper demonstrates that they fail so often that they should be completely disregarded.

(Received August 1971)

The CAM project at LSE is investigating a computer-assisted methodology of developing data processing systems; the project is currently financed by the Science Research Council. The present area of research is the development of prototype software to assist the systems designer in structuring programs and files to satisfy defined data processing requirements on a given computer configuration. While investigating the methodology of systems design, it became clear that various false rules of thumb are widely used in practice; this paper attempts to disprove these rules.

Martin (1967), Losty (1969), Daniels and Yeates (1969) are among the more well-thumbed authors to publish rules based on the hit ratio (or activity ratio) of a file; this parameter measures file activity for a batch-processing run as

$$\text{Hit ratio} = \frac{\text{Number of records accessed}}{\text{Number of records on file}} .$$

The common version is 'if hit ratio is low assign the file to a direct access device, otherwise magnetic tape' and its variations indicate file access method or file processing mode. 'Low' is often left undefined or assigned such apparently arbitrary values as 1, 5, 7 or 10%. Although these rules have probably sold many disc computeis and eliminated much time and thought at the systems design stage, they are completely false as evidenced by the following general and particular criticisms.

Objectives of systems design

In the very first place, what are the objectives of systems design? Generally, a system should be:

1. Economical (ie: cost-effective), compared with alternative systems.
2. Accurate, to ensure that all required operational outputs are correct.
3. Timely, in that these outputs must be produced on schedule.
4. Flexible, to cope with unforeseen requirements and change.
5. Robust, to stand up to variations in workload.
6. Secure, to guarantee regular service in spite of failures.
7. Efficient, so as not to waste valuable resources.
8. Maintainable, and, in particular, intelligible to future generations.
9. Implementable, with due regard to available resources, particularly programming skills.
10. Compatible, with existing systems.
11. Portable, over a range of hardware/software configurations.
12. Acceptable, to any design standards imposed by the organisation.

In practice, these objectives conflict and their relative weights vary between systems and designers; therefore, it is extremely doubtful that any rule of thumb will satisfy these variations. The rules in question supposedly satisfy the efficiency objective (by reducing computer run time); however, not all the remaining objectives will be satisfied, particularly that of economy (which requires balancing a costed saving in computer run time against increased costs of hardware and stationery).

File design decisions

In the second place, what decisions have to be taken in order to fully specify a file? Three have already been mentioned as possessing rules of thumb but Waters (1970) indicates that the full set for key orientated files is:

1. Data content: which data items constitute a record and which records constitute the file (including controls)?
2. Sequence: random or sequential and, in the latter case, which particular sequence?
3. Access method: search, index or algorithm (the simplest being self-indexing)?
4. Format: binary or decimal and fixed or variable length data?
5. Processing mode: sequential, skip (or selective) sequential or random?
6. Security system: generation, dumping or duplication?
7. Breakpoints (or restart points): what dumping is necessary?
8. Device: to which device(s) should the file be assigned?
9. Buffering: single, double or multiple buffering?
10. Block size: standard or optimal?
11. Channelling: which devices should be allocated to which channels?

The first four decisions are program-independent but the remaining seven depend heavily on program organisations. The decisions are listed in the order that they might logically be taken and not in order of significance with respect to computer run time. In practice, the relative significances of these decisions vary between systems: for example, choice of block size might be critical on a small configuration but trivial on a larger one.

Clearly, these file design decisions are highly interactive and research is showing that no decision should be taken in isolation; in fact, the design process must necessarily be iterative. Consequently, it is very doubtful that any rule of thumb will justify a short-cut in this design process.

File activity parameters

In the third place, what parameters fully define the activity of a file? File size could be included here in that, whatever the hit ratio, a small file might be input to core storage, updated and then dumped; further factors that affect systems design are:

1. Hit ratio, defined above; for example, only 1% of customers might order each day or 100% of employees might be paid each week. Hit ratio can vary with:

(*a*) Time; for example, in a seasonal trade 80% of customers might order in each of the initial days of the season whereas less than 1% might order in each of the final days of the season.

(*b*) Message; for example, customer orders might hit 5% of the file whereas customer statements or bills might be output for 60% of the file. In practice, every file is completely hit at some frequency, if only for security purposes when dumping.

A record is but one level of storage within a file, the other levels including block (and/or bucket), track, cylinder and device; each has a hit ratio given by (assuming hit records are randomly spread over the file),

$$\text{Track Hit Ratio} = 1-(1\text{-Record Hit Ratio})^{\text{Number of Records in Track}}$$

and similarly for block, cylinder and device hit ratios. Notice that record hit ratio could be low but track hit ratio could be high; for example, 1% record hit ratio with 100 records per track yields 64% track hit ratio.

2. Hit group, defined as an area of the file containing a significant proportion of the hit records; for example 80% of customer order items might relate to 20% of the product file, this being a hit group of catalogued products. Hit group can vary with:
(*a*) Time; for example, a hire purchase file of clients' records sequenced on monthly repayment day would feature a hit group (in terms of posting repayments) centred on the current date.
(*b*) Message; for example, a public utility system (e.g. electricity billing) might use a file of consumer records sequenced on meter-reading round number. One hit group refers to output messages instructing meter-readers on their rounds, a second hit group refers to input messages of meter readings and possibly a third hit group refers to input messages of customer payments; all these hit groups vary over a time cycle (of usually one quarter).
3. Fan in/out ratio, defined as

$$\frac{\text{Number of input/output messages}}{\text{Number of hit records}};$$

this measures the number of times each active record is hit by input/output messages. This fan in/out ratio can vary with:
(*a*) Time; for example, customer orders might fan in 200 to 1 for each product at the start of the season but only evens at the end of the season.
(*b*) Message; in a banking system, cheque debits might fan in 2 to 1 for each moving current account record whereas overdraft warnings should only fan out evens.
4. Volatility, loosely defined as measures of the file's 'breathing' (i.e. expansion and contraction or growth and decay); usually, this is a measure of the percentage of insertion and/or deletion records over time. Examples of volatility are:
(*a*) A product file might be fairly static if no expansion is anticipated to a well-established product range.
(*b*) A medical research file might have a steady growth rate of 10% per annum, without any decay, if it contains a record for each known sufferer of a particular disease (whether alive or dead).
(*c*) A project control file might have a steady decay rate of 2% per month, without any growth, if it only contains records for outstanding tasks.
(*d*) An airline reservation system file of impending flights might be extremely volatile in that each flight record might exist for only 10 hours so that the file is continually changing without significantly growing nor decaying.
5. Overflow characteristics, loosely defined as measures of any patterns in record insertions; usually, this is an indication of any 'point overflow' conditions that might occur. For example, a rapidly expanding product file might have new product records inserted with consecutive keys (i.e. record identifiers being product code numbers in this case). Generally, files seldom possess this characteristic as insertion records are evenly spread throughout the file.

Thus, hit ratio is only one file activity parameter of many, and often the least significant; consequently, any rule of thumb based exclusively on hit ratio must have doubtful value.

Generally, these parameters are often difficult to estimate and measure and they have a further habit of changing due to the influences of dynamic environmental systems; witness how often an estimated 5% hit ratio at design stage operationally grows to exceed 20% due to increased demand and then reaches 100% due to some unforeseen requirement. Thus, over-reliance on these parameters to achieve efficiency in computer run time can result in a system that is neither flexible nor robust; further, if the computer system's design is heavily based on these parameters then they should be operationally monitored and reviewed to detect any significant changes which would degrade the system's performance.

Other general criticisms

So far, the rules of thumb in question have been attacked for not meeting general design objectives, for oversimplifying a complex design process and for ignoring equally-important design criteria. Further technical criticisms are:

1. It is unlikely that any rule of thumb will cope with the rapidly developing technology of secondary storage devices; hardware performances increase (e.g. magnetic tape transfer rates and exchangeable disc pack capacities) while costs decrease and not proportionally.
2. Any rule of thumb that globally refers to direct access devices is bound to mislead. The timing performances of zero seek time devices (e.g. fixed head drums and discs) are usually far superior to other devices (e.g. movable head drums and discs, datacell and magnetic card files). Even the timing performance of a particular device can significantly vary between configurations due to differing software implementations.
3. Even if the rule refers to a particular device, for example an IBM 2311 exchangeable disc pack, then it should state whether or not the device be dedicated to the file. There can be a marked difference in timing performance for a file exclusively allocated to a device and for the same file sharing the same device in a multi-programming environment (due to interrupted arm movements).
4. Any rule of thumb that globally refers to magnetic tape is bound to mislead in light of the wide range of transfer rates (between 20 and 320 Kc, say).
5. Finally, a distinction should be drawn between a reference file (i.e. input only) and an updated file (i.e. both input and output) as, yet again, timing considerations could be significantly different; for example, updating time could be double reference time and significantly more if read after write techniques are employed to check the recording on direct access devices.

Thus, global rules of thumb tend to beg all the complex timing questions associated with the already wide and developing range of secondary storage devices.

There now follows detailed criticism of the most common rules, each rule being broken by examples.

If hit ratio is low update the file skip sequentially, otherwise sequentially

This version suggests that the file is to be assigned to a direct access device and any messages are to be sorted to the same physical sequence as the file; the choice is apparently between updating hit records *in situ* (by overlaying) and updating the entire file. Superficially, this version of the rule appears to be sensible as unnecessary records are not accessed, but it fails for the following reasons:

1. Blocks of records are transferred and not individual records; records are usually grouped into blocks for the following reasons:

(*a*) Direct access devices impose constraints which can severely reduce the effective capacity of the device. For example, each block might suffer gap and software overheads of say 60 to 100 characters; also, unused capacity less than block size can be wasted at the end of each track and some configurations further restrict block sizes to 500, 1,000, 2,000 or 4,000 characters (these latter configurations often call the block a 'bucket' and the term 'block' defines a particular storage capacity of 500 characters). Consequently, records are often packed into blocks to efficiently utilise direct access device capacity.

(*b*) Some organisations impose general standards on block size, say 1,000 characters per block, to achieve configuration and/or device independence.

(*c*) Although a file might possess a low hit ratio for its most frequent updates, it still needs to be totally processed from time to time (if only for dumping purposes as noted above). This sequential processing prefers large block size, subject to core storage availability, to avoid excessive time overheads with respect to rotational delays; Waters (1971) develops an algorithm for this case. This imbalance between small block size for low hit ratio updates and large block size for high hit ratio updates is usually resolved by choosing some intermediate block size.

Thus, since records are usually grouped into blocks, unrequired records will be accessed; further, if

$$\text{Hit ratio} > {}^{1}/\text{Number of records in block}$$

then the vast majority of records will be accessed anyway.

2. Track hit ratio is often more significant than record hit ratio for timing purposes. For example, if the majority of tracks are hit and each hit track contains more than two hit records, then sequential processing is often faster than skip-sequential processing (particularly if block size is the maximum of track size); this is largely due to savings in rotational delays. These conditions can apply even when the record hit ratio is low.
3. Often, any time saved by skip sequential processing is insignificant for small to medium-sized files; for example, sequentially accessing all blocks of a file held on an IBM 2311 exchangeable disc pack could take as little time as 1 minute (particularly if block size is the maximum of track size).
4. Skip sequential processing infers single-buffering the file which, in turn, infers a loss of simultaneity between input, processing and output functions. Double-buffering a sequentially processed file can achieve this simultaneity which again narrows the time gap between the two processing modes. Occasionally, multiple-buffering a sequentially processed file can virtually eliminate the overhead transfer time of inactive records if the processing time of active records is sufficiently high.
5. If the low hit ratio situation is such that all hit records form an isolatable hit group then processing time can be saved by extracting this hit group into a separate file which is processed sequentially. This technique can also apply when a minority of hit records are outside the hit group if less frequent updating of these records is acceptable.
6. The previous five points have demonstrated that even if hit ratio is low there are many cases where sequential processing is preferable. Conversely, if hit ratio is high then it might well be preferable to process the file skip sequentially; for instance, if skip-sequential processing requires less secondary storage capacity.

If hit ratio is low update the file randomly, otherwise sequentially

This version suggests that the file is to be assigned to a direct access device; the choice is apparently between randomly updating hit records *in situ* and updating the entire file. This rule often fails because:

1. Random processing infers the entire file be on-line to the central processing unit; further, if fast restart recovery procedures are necessary, then incremental dumping of original file records can be necessary which demands further secondary storage. These two requirements often cannot be met due to limited on-line storage capacity.
2. Splitting the file into hit and non-hit (or infrequently hit) sections and processing sequentially, as above, might be justifiable; particularly if fan in/out ratios are high which causes multiple accesses to the same records when randomly processing.
3. These previous two points indicate that sequential processing is again often preferable for low hit ratio situations. Conversely, random processing might be used in a high hit ratio situation; for instance, if messages cause a high degree of interaction between separate files then one-shot processing might be required whereby a batch of messages update one file sequentially and remaining files randomly.

Finally, the two rules considered above lead to a comparison between skip sequential and random processing; an extra consideration is the time to sort messages to the file sequence compared with the seek time incurred by random processing. Often, seek time exceeds sort time in which case skip sequential processing might be preferable; even some quick response systems sort message queues to file sequence for this reason.

If hit ratio is low use indexed sequential file organisation, otherwise sequential

This version suggests that the file is to be assigned to a direct access device and confuses the distinction between file access method and file processing mode. The above comments have discussed the choice of processing mode and the following points complete the criticism of this rule:

1. If the file has a severe point overflow characteristic then indexed sequential organisation could cause high inefficiencies when accessing the file. Thus, whatever the hit ratio, sequential, indexed random or algorithmic organisation would probably be chosen.
2. If the file is highly volatile then similar inefficiencies might result from indexed sequential organisation. Whatever the hit ratio, another organisation would probably be chosen.
3. If the hit ratio is low and record keys are dense (i.e. relatively few values of the range are unassigned), then the simplest algorithm (self-indexing) might be preferable to indexed sequential organisation.
4. On the other hand, if the hit ratio is high then indexed sequential organisation could be chosen to support a future quick response requirement.

If hit ratio is low use disc, otherwise magnetic tape

This version of the rule is the most general (other than replacing 'disc' by 'direct access device') and embodies confusion between file organisation methods, file processing methods and devices; the first two decisions have been discussed above, therefore it merely remains to compare disc and tape for sequential processing purposes, as follows:

1. Discs are faster than most tapes (other than high-speed varieties) when sequentially processing; thus, a tight time requirement (e.g. fast turnaround or fit heavy existing computer workload) might favour discs.
2. Sequentially processing discs might require fewer devices than a similar tape system as many files can be assigned to a

disc; further, the 'split cylinder' technique can be used to reduce time-consuming seeks in this case. For example, a two-disc configuration might match the throughput of a five-tape configuration (but possibly might still be the more expensive of the two).

3. Even sequential disc files have the flexibility of direct access when required, if only by the 'binary chop' technique; this facility might save passes of the file. For example, amendments to record keys can require that the file be resequenced in the current run; a tape file will require an extra pass to achieve this, whereas the direct access facility can be used to avoid this for the disc version.
4. There are many more arguments that favour disc sequential processing but, on the other hand, tape sequential processing often (but not always) proves to be more economical. Generally, a costed saving in computer run time must be weighed against increased hardware and stationery costs.

Conclusion

The rules of thumb have been disproved. A methodology of computer systems design must necessarily commence with careful consideration of the objectives and continue through an iterative, decision-making procedure which attempts to meet these objectives; the CAM project is developing such a methodology. It appears doubtful that any short-cuts can justifiably be taken to achieve an efficient design other than stopping the design procedure when an acceptable solution has been achieved.

Finally, the author wishes to acknowledge the assistance of his colleagues at LSE, particularly Mr. F. F. Land and Dr. A. H. Land of the Statistics, Mathematics, Computing and Operational Research Department. The author would be pleased to discuss the details of this paper with any interested organisation, particularly in relation to specific computer systems design problems.

References

Daniels, A., and Yeates, D. (1969). *Basic Training in Systems Analysis*, London: Pitman (for National Computing Centre).
Losty, P. A. (1969). *The Effective Use of Computers in Business*, London: Cassell.
Martin, J. (1967). *Design of Real-Time Computer Systems*, New Jersey: Prentice-Hall.
Waters, S. J. (1970). Physical Data Structüre. Paper 6 of Proceedings of BCS Conference on Data Organisation.
Waters, S. J. (1971). Blocking Sequentially Processed Magnetic Files, *The Computer Journal*, Vol. 14, pp. 109-112.

Correspondence

To the Editor
The Computer Journal

Sir,

High level languages are unnecessarily complex for the inexperienced user, D. G. Evershed and G. E. Rippon say in a paper in Vol. 14, No. 1, 1971 of this *Journal*. The authors of the paper discuss a number of modifications to FORTRAN, ALGOL and other high level languages. The goal of most of the suggested modifications is to make it easier to write programs which are acceptable to the compiler. Many of the suggestions made by the authors are very valuable. However, the authors do not seem to be aware of a basic conflict underlying some of their suggestions.

The basic conflict is the following: The errors in a computer program can be divided into two categories:

1. Language errors, which can be detected by the compiler.
2. Logical errors, which can only be detected by erroneous results during the execution.

The second category, the logical errors, are much more troublesome than the language errors. The reason for this is that the compiler gives a good and useful diagnostic for most language errors, which makes it very simple for the programmer to correct his errors. For logical errors, on the other hand, the process of finding and correcting them is often much harder. There is even a large risk that logical errors are not discovered until production use of the program has begun. Sometimes, logical errors are not discovered at all, which means that the results of the runs of the program may be dangerously false.

Because of this, it is much more important to design a computer system which gives few logical errors than to design a system which gives few language errors.

In many cases, a programming language construct can be designed either to give few language errors or to give few logical errors. The reason for this is that the language can be designed in such a way that as many common logical errors as possible will also cause language errors. If a programming language is designed in this way, then the compiler has much larger possibilities to help the programmer avoid logical errors.

A very simple example; the following ALGOL program contains an error:

```
integer abcde;
.....
abcde := abode + 1;
```

The intention of the programmer was to increase *abcde* by 1. However, by mistake, he instead wrote *abode* on the right hand side. This may cause a logical error which is very difficult to discover, if the sequence above was written in FORTRAN. However, in ALGOL, a language error occurs: *abode* is an undeclared variable. Thus, the compiler can detect the programming error in ALGOL but not in FORTRAN.

A second example; the following construct is allowed in both ALGOL 60 and FORTRAN but not in ALGOL 68:

```
real a; integer i;
.....
i := a;
```

The difficulty with this statement is that real variables can be converted into integers in many different ways. You can make a correct rounding (using different ways of rounding) or you can take the nearest lower integer, either with sign (as the ALGOL *entier* function) or without sign (as the FORTRAN *int* function). In fact, the program above will be executed in one of these ways in ALGOL 60 and in another way in FORTRAN. A very common programming error is to assume one conversion when the real conversion is another than the one assumed. This error cannot occur in ALGOL 68.

A third, more complex example, is the handling of pointer variables in various languages. If these pointer variables are typedeclared and typechecked (like in ALGOL 68 and in Simula 67) then the risk of undetected logical errors is much smaller than without such checking (like in PL/1 or in SIMSCRIPT). Also, a garbage collector (like in LISP, SNOBOL, ALGOL 68, SIMULA 67) gives smaller risk of programming errors than explicit deallocation of list structure records (which is done in PL/1 and SIMSCRIPT).

(Continued on page 36)