JANUARY/FEBRUARY 1984
Volume 12 Number 1

I.P. Sharp newsletter

technical supplement 48

Making Things Fit, Part 3

How big should a file component be?

The question of how big a file component should be really addresses two issues:

1) What is an optimal size?
2) How can data be segmented to achieve this optimal size?

An important factor in answering this question is that the cost of reading a component grows more slowly than the size of the component. For example, the CPU time required to read 100 components containing ⍳5000 is only three times the time required to read the same number of components containing ⍳500, even though ten times as much data is being moved. Thus, we learn that using *large components* instead of small ones can reduce execution time.

The size of the active workspace is the real limiting factor on the size of file components. A good rule of thumb (and by no means infallible) is: your method of segmenting data should ensure that no component exceeds ten per cent of the size of a clear workspace. This means that in a 350K workspace, no component should ever exceed 35K bytes. This leaves a reasonable amount of room for other variables and functions in the workspace. While ten per cent may be the maximum, five per cent is a more reasonable average. This allows for some growth, which is important.

The two major ways to split data into file components correspond to the two types of data—uniform and non-uniform. With uniform data, every scalar in an array represents a fact. These facts are numbers, and they all have the same unit of measure. For example, a 12-element vector of monthly budget figures in dollars is uniform data.

With non-uniform data, a collection of scalars represents a fact. The scalars can be numbers or characters. The numbers may be measurable quantities or identifiers. The scalars may be grouped into units of meaning called **fields**, and an entire fact is called a **record**. This type of data is non-uniform because the data types of the scalars may differ, as well as the unit of measure (if any) of the numbers. For example, a vector consisting of a department number, account number, dollar amount, and date is a record of non-uniform data.

How do you segment uniform data? Consider a typical APL application. A company wants to monitor the performance of its sales organization. It has monthly statistics by sales unit by product for several years, containing both the sales plan and the actual figures. This can be treated as a five-dimensional array: Year by Type (plan, actual) by Unit by Product by Month. If there are a lot of units or products, the data won't fit in the workspace.

To make the data fit, you must split off some of the dimensions. If there are 50 units and 200 products, you might make each file component a matrix in which the rows represent products, and the columns represent months. Each matrix will be 200 by 12. If the data is stored in four-byte integers, the component will take approximately 9600 bytes, which is a reasonable size. There will be a matrix for each Year/Type/Unit combination. For five years of data, that will mean 5×2×50 = 500 components.

To locate the component associated with a

given combination, you can compute the following:

```
1+(0,NUMBER∆OF∆TYPES,NUMBER∆OF∆UNITS)⊥
  ¯1+YEAR∆INDEX,TYPE∆INDEX,UNIT∆INDEX
```

You may want to provide for one or more of the dimensions becoming longer (a very likely possibility). In this case, use the number computed above to index into a vector that indicates which component relates to that combination. When a dimension becomes longer, you will have to insert elements in the vector to represent the new data, but the components on file can remain where they are.

There are no hard-and-fast rules for splitting dimensions to obtain an optimal file design. Listed below are some helpful guidelines:

1) Keep data that will be used together most frequently in the same file component to minimize file operations
2) Split as few dimensions as possible to minimize looping
3) Provide an orderly means for growth to minimize maintenance headaches

How can non-uniform data be segmented? In *Technical Supplements 40* and *41*, six methods of searching files containing such data are described:

1) Sequential search of a random file
2) Sequential search of an inverted file
3) Binary search of a sorted file
4) Directory search of a sorted file
5) Sequential search of a buffered (random or sorted) file
6) Algorithmic search of a hashed file

These six methods reduce to four file organizations: random, sorted, inverted, and hashed. The following discussion explains how to group records under each scheme in order to prevent *WS FULL* and maximize performance.

Managing a file in which the records are not stored in any special order is not very hard. Records are blocked together in components. The component size should be between five and ten per cent of □*WA* in a clear workspace. New records are appended to the bottom of the last component. When it is filled, a new component is appended to the end of the file. The following function shows how this works.

```
    ∇ RECORD APPEND∆RANDOM TIE
      ;BLOCK;PTR
[1]   PTR←¯1+1↑1↓□SIZE TIE
[2]   BLOCK←□READ TIE,PTR
[3]   →((1↑ρBLOCK)<MAXBLOCKSIZE)ρAPPEND
[4]   BLOCK←(0,¯1↑ρRECORD)ρ0
[5]   PTR←PTR+1
[6]   BLOCK □APPEND TIE
[7] APPEND:BLOCK←BLOCK,[1] RECORD
[8]   BLOCK □REPLACE TIE,PTR ∇
```

Managing a file of sorted records takes a bit more effort. If the file is permanently sorted, a directory can be created to find records more easily. Records are blocked together in components with the other records that have the same identifying field (sort key). But since the distribution of records over the different keys may not be even (and probably isn't), a means of segmenting the data further must be used.

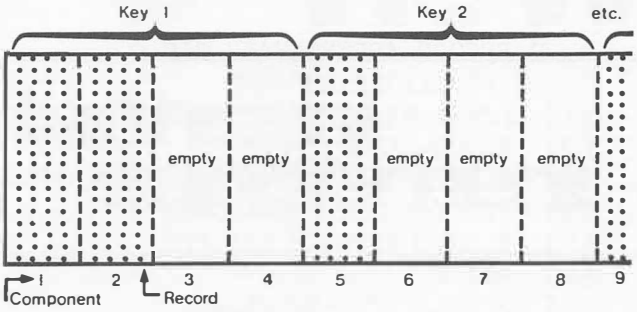
The solution is to allow for multiple components for each key. The directory must contain three columns:

1) the identifying field
2) the first component containing records that are identified
3) the last component containing records for that identifier.

Thus records are first split by key, and then again into blocks. The maximum block size should be five to ten per cent of □*WA* in a clear workspace, divided by the number of bytes each record needs.

A group of components is reserved for each key. The number of components reserved should be several times greater than the number of components you expect a single key to require. For example, if the average key will require one block of records, allocate five or more components per key. The extra components are reserved by appending empty arrays. The empty components aren't read unless they are needed, and so cost nothing. Empty components reserved for future use are a cheap way to allow for growth. The following diagram shows the layout of such a file structure.

. . . Making Things Fit



SORTED RECORDS BLOCKED BY RECORD

The following program shows how a new record is appended to such a file structure. Note that it attempts to place the record in the first block which isn't full, since deletions may have eliminated records. It also contains a paranoia check, just in case all your extra empty components have been used up.

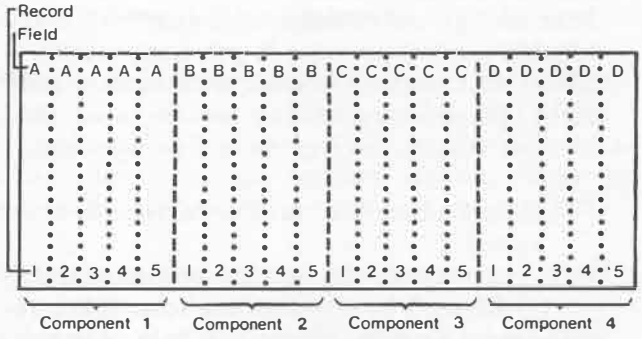
```
    ∇ RECORD APPENDΔDIRECT TIE
      ;BLOCK;ROW;FIRST;LAST
[1]   ROW←INDEX[;1]⍳RECORD[KEYFIELD]
[2]   CTR←INDEX[ROW;2]-1
[3]   LMT←INDEX[ROW;3]
[4]  LOOP:→(LMT<CTR←CTR+1)⍴END
[5]   BLOCK←⎕READ TIE,CTR
[6]   →((1↑⍴BLOCK)≥MAXBLOCKSIZE)⍴LOOP
[7]   BLOCK←BLOCK,[1] RECORD
[8]   BLOCK ⎕REPLACE TIE,CTR
[9]   →EXIT
[10]END:→(CTR>BLOCKSPERKEY)⍴FULL
[11]  BLOCK←(1⌈¯2↑⍴RECORD)⍴RECORD
[12]  BLOCK ⎕REPLACE TIE,CTR
[13]  INDEX[ROW;3]←CTR
[14]  →EXIT
[15]FULL:'NO SPACE FOR KEY ',⍕INDEX[ROW;1]
[16]EXIT: ∇
```

An inverted file structure segments its data by field, rather than by record. The simplest inverted structure puts all of the values of one field from all records in a separate component. If there are several thousand records in the file, the components will get too big to fit in the workspace. So, the data is first split by field, and then each field is split into blocks. The block size should be five to ten per cent of ⎕WA in a clear workspace, divided by the number of bytes one field entry takes.

There are two ways to lay out an inverted file:

1) All components which are part of a single field can be stored contiguously. If this structure is chosen, room for growth can be provided by putting empty components at the end of every group of components which comprise a single field. Such an organization must be reorganized when all of the empty components are filled.
2) All components which contain a given set of records can be stored contiguously. If this structure is chosen, no special provision for adding records need be made. When a block is filled up, a new one is started. However, such an organization must be reorganized when a field is added or deleted.

The following diagram shows the layout of an inverted file.



INVERTED RECORDS BLOCKED BY FIELD

The next program shows how a new record is appended to an inverted file. Note that it has to do a ⎕READ and ⎕REPLACE for each field in the record. While inverted files are fast to search, they are slow to update.

```
    ∇ RECORD APPENDΔINVERT TIE
      ;FLDCTR;FLDLMT;BLKCTR;BLKLMT;BLOCK;BASE
[1]   FLDCTR←0
[2]   FLDLMT←⍴RECORD
[3] FLDLOOP:→(FLDLMT<FLDCTR←FLDCTR+1)⍴FLDLOOPEND
[4]   BASE←BLOCKSPERFIELD×¯1+FLDCTR
[5]   BLKCTR←0
[6]   BLKLMT←BLOCKSPERFIELD
[7] BLKLOOP:→(BLKLMT<BLKCTR←BLKCTR+1)⍴BLKLOOPEND
[8]   BLOCK←⎕READ TIE,BASE+BLKCTR
[9]   BLOCK←BLOCK,RECORD[FLDCTR]
[10] BLOCK ⎕REPLACE TIE,BASE+BLKCTR
[11] →BLKLOOP
[12]BLKLOOPEND:→FLDLOOP
[13]FLDLOOPEND: ∇
```

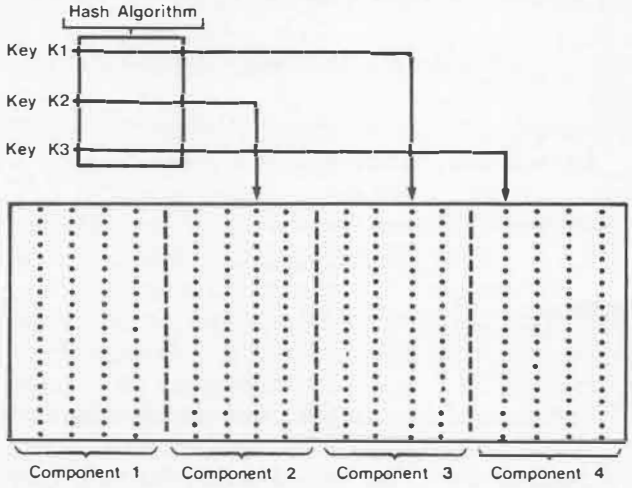
A hashed file contains non-uniform data stored by record. The records are neither in random order, nor sorted order. Instead, they

are scattered among a limited set of components based on an arithmetical calculation done on the identifying field. The calculation should distribute the records over the available components relatively evenly. A maximum block size should be established as with the sorted file.

What happens when too many records map into the same component? There is a fixed number of components used for the hashed records, and any beyond them are used as an overflow area. Records which don't fit in the component they are mapped to are stored in an overflow area, in sequential order. Obviously, if a significant portion of the records go to the overflow area, this method will be even slower than the sequential method to update and to search. The key to this method is to have enough components available, with a reasonable number of records per component. Since most hashing is done with modulo arithmetic, properly segmenting records using this method usually involves the following steps:

1) Approximate the number of records to be stored
2) Approximate the number of components to be used by dividing the total space required to store all records by a reasonable component size
3) Find a hash number (usually prime) which distributes the anticipated records relatively evenly over the components

The next diagram shows how records are mapped into components under this scheme.



HASHED RECORDS

The following program shows how a record is appended to a hashed file.

```
    ∇ RECORD APPENDΔHASH TIE
      ;BLOCK;CTR;LMT;PTR
[1]   PTR←1+HASHNUM|RECORD[KEYFIELD]
[2]   BLOCK←⎕READ TIE,PTR
[3]   →(MAXBLOCKSIZE≤1↑ρBLOCK)ρOVERFLOW
[4]   BLOCK←BLOCK,[1] RECORD
[5]   BLOCK ⎕REPLACE TIE,PTR
[6]   →EXIT
[7] OVERFLOW:CTR←HASHNUM
[8]   LMT←¯1+1↑1↓⎕SIZE TIE
[9] LOOP:→(LMT<CTR←CTR+1)ρMORE
[10]  BLOCK←⎕READ TIE,CTR
[11]  →(MAXBLOCKSIZE≤1↑ρBLOCK)ρLOOP
[12]  BLOCK←BLOCK,[1] RECORD
[13]  BLOCK ⎕REPLACE TIE,CTR
[14]  →EXIT
[15]MORE:BLOCK←(1⌈¯2↑ρRECORD)ρRECORD
[16]  BLOCK ⎕APPEND TIE
[17]EXIT: ∇
```

To decide which of these four methods to use in segmenting non-uniform data depends on the way the data will be used. The last three can all run quite efficiently under the proper circumstances. The suggestions in this section will make it possible for you to segment the data so that it fits in the active workspace.

Conclusion

The chief way to make data fit in a file is to design a flexible file structure. Such a flexible structure provides room for growth in the volume of data stored, as well as for variations in the distribution of the data over the identifiers. If you design files with these goals in mind, you will provide applications which serve their users well.

Acknowledgements

My thanks to J. Henri Schueler, Peter Wooster, and Gordon Ross for their help with this article. ■

Robert Metzger, Rochester