DISCUSSION PAPER NO. 84

COST EVALUATION OF STORAGE SCHEMES

by

Jair M. Babad, V. Balachandran
and
Edward A. Stohr

May 10, 1974

COST EVALUATION OF STORAGE SCHEMES

by

Jair M. Babad[*], V. Balachandran[#] and Edward A. Stohr[#]

[*]Graduate School of Business, University of Chicago
5836 S. Greenwood Avenue, Chicago, Illinois 60637
Tel.: 312-753-3601

[#]Graduate School of Management, Northwestern University
Nathaniel Leverone Hall, Evanston, Illinois 60201
Tel.: 312-492-3603

Suggested SIG: SIGIR Alternative SIG: SIGBDP

ABSTRACT

In this paper we present a methodology for the cost evaluation of file system performance. The cost structure we consider takes into account the various operations that are required during the processing of data in the system. The cost evaluation approach is then applied to several systems. A new storage scheme--a partially ordered file--is proposed, and experimental data which demonstrate its performance are presented. Finally, the cost evaluation approach is applied to this proposed storage scheme.

COST EVALUATION OF STORAGE SCHEMES

by

Jair M. Babad, V. Balachandran and Edward A. Stohr

1. Introduction

In this paper we present a methodology for the cost evaluation of file system performance. The demands on a file system consist of four basic data processing operations: insertion and deletion of records, searching for a specific record, and sequential processing of the file in some predetermined order. Sequential processing is usually preceded by a sorting operation according to the required order. Accordingly, an evaluation of a file system has to take in account the cost of insertions, deletions, searching, sorting and sequential processing, as well as the frequency of these operations. These costs depend on the organization of the file and the processing algorithms used; however, the file organization will usually determine which algorithms should be selected. As a result, we can associate a cost structure with each file organization. Furthermore, the cost of the sequential processing is independent of the file organization, once the records are sorted in the desired order. Therefore, this cost may be ignored when the cost performance of the system is considered.

Many authors discuss the evaluation of storage schemes; see Katter [2] or Lancaster [4] for detailed reviews and references. However, the "micro" approach we present here, which goes down to individual operations on the file system is hardly ever encountered. One such example is given by Lefkovitz [5] whose analysis is concerned with list structures on disk devices. A similar approach is found in Babad [1] where the costs of storage and different storage devices and of individual operations on the file system are minimized by a partition of the file into subfiles and subrecords.

An appropriate cost structure, and the underlying assumptions, are described in Section 2. The model might be applied to many file organizations, but for expository purposes we limit our discussion to sequential file systems, e.g., file directories or tape systems. The usual sequential file organization, in which the information is always stored in some predetermined order, is analyzed in Section 3. An alternative organization, which utilizes partial ordering, is described and examined in Sections 4 and 5. In this scheme, the records are stored as a heap (Knuth [3]). Since this is an unusual storage scheme we also describe the processing algorithms, and include results of some performance tests.

2. The Cost of Data Processing

As was mentioned above, four basic operations have to be taken into account: insertions, deletions, searches and sorting (sequential processing). These will be denoted by the subscripts i, d, s and p, respectively. Let n_x be the average number of type x operations per time period, and c_x the average cost of one type x operation. c_x might be measured in many ways, e.g., number of records accessed before the operation is completed, cost in monetary units, time, etc. For simplicity, we chose to measure it as the average number of records accessed, since the required processing time and monetary expenses are directly related to this measure. In addition, we assume that the average size of the file, i.e., the number of records N, is relatively stable. The costs c_x are clearly dependent on the size of the file, but due to this assumption we may suppress this dependence in the following discussion.

Finally, we assume that requests for service from the file system are satisfied individually as they arrive; as a result, insertions, deletions, and searches in our cost equation pertain to individual records.

Using this notation, we get as the average processing cost per time period

$$C = n_i c_i + n_d c_d + n_s c_s + n_p c_p \tag{1}$$

The cost coefficients c_x are dependent on the file organization, and our aim is to find a file organization that would minimize the total expected cost C, subject to the exogenous data as given by the n_x's.

It is worth noting that the need for sorting depends heavily on the order in which the file is kept. Specifically, if a sequential processing has to be done in the same order in which the file is usually maintained, there is no need for sorting. Therefore, we assume that one specific order is the main ordering for sequential processing, and we denote by n_m the average number of sequential processing requirements according to this order. Accordingly, we also denote by n_a the average number of other sequential processing requests during a time period. However, some updating processing might be needed even for the n_m requirements, as is illustrated in the next section. Therefore, we replace the c_p cost value by c_m and c_a, which are the costs associated with n_m and n_a respectively. Equation (1) is then rewritten as

$$C = n_i c_i + n_d c_d + n_s c_s + n_m c_m + n_a c_a \tag{2}$$

3. The Cost of Sequential Processing

In order to illustrate the usage of the cost structure as described above, we apply it to sequentially stored data. For example, such data might be the directory, or index, of a file system. Such a directory is usually ordered according to some key, and major efforts are invested in keeping it updated; thus, insertions and deletions result in considerable processing. Several approaches for the updating problem are known, and some will be described below; we, however, assume that indirect addressing or linkage schemes are not used.

Consider first the case when the directory is updated by insertions in place and by physical deletion of deleted records. As a result, deletions and insertions entail shifting of records. On the average, half the directory has to be shifted for each insertion or deletion (assuming that these operations pertain to records whose key is uniformly distributed in the directory). This shifting thus requires N operations -N/2 for getting a shifted record, and N/2 for putting it in its new place. In addition, a search is associated with insertions and deletions, in order to find the location in which the record has to be inserted or the deleted record. Thus

$$c_i = c_d = c_s + N$$

The cost c_s of searching depends on the searching algorithm that is employed, and this in turn depend on the storage media on which the directory is stored. If the directory is stored on a sequential device, a sequential search is needed, and this takes on the average N/2 accesses. On the other hand, if random access is available, binary search might be used and c_s is reduced to about Log N (logarithm in base 2 is assumed throughout this paper). As for sorting, we note that $c_m = 0$, while c_a is proportional to N log N, with a proportionality constant k, say. Summing up all the cost elements according to (2) we get as the expected total cost

$$C_{1s} = N(3n_i + 3n_d + n_s + 2kn_a \text{Log} N)/2 \tag{3}$$

for a sequential directory on a sequential storage device, and

$$C_{1r} = N(n_i + n_d + kn_a \text{Log}N) + (n_i + n_d + n_s)\text{Log}N \tag{4}$$

for a sequential directory on a random access device. As is clear from the context, $C_{1r} < C_{1s}$.

Another strategy for updating the directory marks the deleted records, rather than deleting them physically. In this case, prior to any sequential processing the directory is "cleaned" by copying the unmarked records into a new directory. In this case, the file size changes from time to time. Specifically, it is N immediately after a sequential processing, and it increases, by the number of deleted records, till the time of the next sequential processing. Assuming uniform processing requirements over the time periods, the number of deleted records between two consecutive sequential processings is $n_d/(n_a + n_m)$. Therefore sequential processing is preceded by the reading of $N + n_d/(n_a + n_m)$ records and the rewriting of N records into the "clean" directory, which might later be sorted. In other words, c_a and c_m as given above have to be incremented by $N + N + n_a/(n_a + n_m)$. Similarly, the file size for searching and insertions is on the average $N + n_d/2(n_a + n_m)$. The cost of deletion has to be modified into $1 + c_s$, of which c_s is attributed to the search for the record to be deleted, and the deletion itself is done by one writing operation of the marked record. Summing up all the cost elements of the equation (2), we get after some algebraic manipulation that the expected total cost is

$$\begin{aligned} C_{2s} &= N(3n_i + n_d + n_s + 4n_a + 4n_m + 2kn_a\text{Log}N)/2 \\ &\quad + n_d(3n_i + n_d + 1 + n_a + 4n_a + 4n_m)/4(n_a + n_m) \end{aligned} \tag{5}$$

for a directory on a sequential storage device, and

$$\begin{aligned} C_{2r} &= N(n_i + kn_a\text{Log}N + 2n_a + 2n_m) + (n_i + n_d + n_a)\ \text{Log}(N + n_d/2(n_a + n_m)) \\ &\quad + n_d(n_i + 1 + 2n_a + 2n_m)/2(n_a + n_m) \end{aligned} \tag{6}$$

for a directory on a random access device. As before, we know from the context that $C_{2r} < C_{2s}$. A direct comparison of C_{1r} and C_{2r}, for example, is notationally hard. However, once the exogenous data of N and the n_x's is known, the choice of the updating strategy is immediate.

Other updating strategies might be used. For example, insertions might be put at the end of the directory, which is updated and rewritten every time a sequential processing is done. Alternatively, rewriting of the directory might be deferred and be done less frequently. Similarly, insertions and deletions might be accumulated, and the directory updating in this case consists of merging the old directory with the batched transactions. In this case, though, searching might yield erroneous results, since the file is not updated. Or, several of these strategies might be combined. The cost equation can be easily written for every given strategy, and the most appropriate strategy can be chosen for given exogenous data. However, we will not discuss this file organization further here.

4. A Partially Ordered File - Description and Experimental Results

In the preceding section we saw that the costs of insertions and deletions are relatively high. Since these operations are of the utmost importance if the file has to remain updated, a strategy which reduces the costs of insertions and deletions seems attractive. From the discussion in the last section it is clear that the main obstacle for such a strategy is the ordering restriction that was imposed on the file. Thus, a storage scheme in which the file is partially ordered suggest itself. Clearly, some tradeoffs exist between the processing of ordered and partially ordered files. In a partially ordered file, sorting, or partial sorting, is always needed; thus the costs, c_m, associated with n_m go up, relative to an ordered file. As for c_a, it is the same for both storage schemes, because in either case a full sorting is required. In addition, the searching costs in a partially ordered file might increase, relative to an ordered one, since binary search cannot be used without a full underlying order. On the other hand, a judiciously chosen partial order might considerably reduce the cost of insertions and deletions. Thus, for many combinations of exogenous data of N and n_x's, a partial order will be preferrable.

The partially ordered storage scheme which we propose is the heap structure (Knuth [3]). Such a storage scheme might be described as a balanced binary tree, in which the value that is stored at the root of any subtree is greater than (or equal to) the values that are stored in other nodes of the subtree. The heap structure, in contrast to many other tree schemes, may be used without any additional links, with consequent savings in storage. Specifically, the "sons" of a root have the indices 2 x index (root) and 2 x index (root) + 1. In addition, the construction of a heap scheme, and its conversion into an ordered file, can be done rather fast. In particular, insertions into a heap take at most LogN data shifts and comparisons, while sorting is proportional to NLogN (Knuth [3]). We also would expect that deletions in a heap are proportional to LogN.

In order to confirm these conjectures and to derive numerical relations, we experimented with files of various sizes. For this experiment, we prepared a subroutine which manipulated the heap structure; specifically, this routine inserted, deleted and searched for records, and also sorted the heap into an ascending ordered file (see the Appendix for details). Various heaps were constructed randomly, with sizes varying from 1000 to 6000 records. Several thousand operations of of deletions, insertions and searches were randomly performed on each heap. For each operation, we measured the number of index comparisons, data value comparisons, and data shifts. The results are discussed in the following paragraphs

Search: Since the data is not ordered, it seems plausible to use a linear search, which requires about N/2 data comparisons. However, we might utilize the heap structure, and search the tree in a more efficient manner. We chose the following strategy: search the root, then the subtree with the smallest son as a root, and then the remaining subtree; such a search was accomplished with the aid of a stack of size [LogN] + 1 (where [x] is the largest integer not exceeding x). A regression analysis revealed that

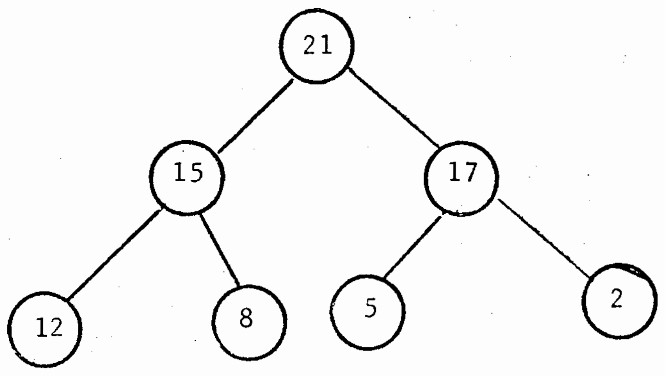

Figure 1: An example of a heap structure

$$c_2 = \text{number of value comparisons} = \underset{(0.0175)}{0.4245N} - \underset{(57.20)}{20.69} \qquad (7)$$

with coefficient of correlation of 0.9916 where the numbers in parentheses are the standard deviations of the coefficients.

Insertions: Using the standard bubbling operation of the first phase of the heap sort (Knuth [3]), we found that the number of index comparisons, value comparisons and data shifts were essentially constant (independent of N). The results we got were:

Table 1 - Insertions

	mean	standard deviation
index comparisons	1.25	0.0129
value comparisons	0.751	0.0113
data shifts	0.751	0.0117

These results are easily explained. About half of the heap elements are stored in the lowest level of the tree, and about three quarters are stored in the two lowest levels; thus, we expect that an insertion will store a new record in the lowest level in about half of the cases, and in the two lowest levels in about three quarters of the possible cases (with similar observations for higher levels). This is true for any value of N, and therefore we would expect that these measures will be independent of N.

Deletions: Our strategy for deletions is as follows. Once a to be deleted record has been found (after a search), we insert in its place the last element of the file. We then check whether we have to bubble it up (as in the insertions case) or to bubble it down (as in the second phase of the heap sort). As was the case with insertions, and for similar reasons, we would expect that the measures for bubbling up or down would be independent of N. For bubbling up we obtained the same results as for insertions, while for bubbling down we obtained

Table 2 - Deletions

	mean	standard deviation
index comparisons	2.15	0.0148
value comparisons	1.36	0.0270
data shifts	0.82	0.0134

Notice that these measures differ from those for insertions, and especially from those for comparisons. This is easily explained: while for insertions (and bubbling up) the record is compared with its "father", we compare the record in the deletions (and bubbling down) case with its sons.

Sorting: For the partially ordered heap scheme, a full order might be achieved by performing the second phase of the heap sort, as given by Knuth [3]. As Knuth asserts, all the operations are proportional to NLogN. Our results are given in the table below:

Table 3 - Partial Sorting Regression

	Coefficient of NLogN	Constant
index comparisons	0.95251	-181.2
	(0.0028)	(111.97)
value comparisons	1.6188	-1368.0
	(0.0075)	(299.39)
data shifts	0.87496	-443.9
	(0.0029)	(116.03)

(the values in parentheses in this and the next table are the standard deviations of the coefficients); all regressions had 1.0 as coefficient of correlation).

For a full sort, as required for the n_a cases, we obtained

Table 4 - Full Sorting Regression

	Coefficient of NLogN	Constant
index comparisons	1.2046	696.3
	(0.0069)	(273.58)
value comparisons	1.7933	-833.5
	(0.0040)	(159.05)
data shifts	1.0496	94.039
	(0.0019)	(76.86)

Our results may be compared with Loeser's [6] observations. He measured the performance of various sorting routines, among them TREESORT3 which is similar to the heap sorting routine. He counted the (value) comparisons, stores and fetches. The results we got for value comparisons with his findings for large random arrays. However, while we got the same coefficient of NLogN over the whole range of tested files, his coefficients for smaller files are smaller than ours. Our results for data shifts have to be compared with his values for "stores". We got consistently smaller values, mainly because our routine was optimized in this respect. Again, our values were consistent on all the tested range of file sizes, in contrast to Loeser's results.

5. The Cost of Processing a Heap File

Using the results of Section 4, and the formulations of Section 2, we get the following cost values for a heap storage scheme:

c_i = 0.751 x 2 = 1.52 (we multiplied by 2, since each data shift involves both read and write operations).

c_d: here we have to take into account the cost of searching, as well as the costs of data shifts both for bubbling up and for bubbling down. But, the values for data shifts in both cases are quite similar--0.75 and 0.82 Being conservative, we employ the higher value, yielding $c_d = c_s + 0.82 \times 2 = c_s + 1.64$.

c_s: we notice that the constant in the regression equation (7) might be ignored, and so c_s = 0.424N (we do not have to multiply here by 2, since searching involves reading only).

c_m: 2 x 0.875 NLogN = 1.75 N Log N from Table 3 (again, we ignore the regression constant).

Similarly $c_a = 2 \times 1.05$ N LogN = 2.1 N Log N from Table 4. Thus, the total expected processing cost for one period is

$$C = (1.52\, n_i + 1.64\, n_d) + N\,[0.425(n_d + n_s) + (1.75\, n_m + 2.1\, n_a)\, \mathrm{LogN}] \qquad (8)$$

We note that the factor k that was used in equations (3)-(6) is equal to 2.1, as was derived for c_a above. It is hard to compare the results of (8) with the former results; however, for any given exogenous data such a comparison can easily be made.

6. Conclusion

In this paper we presented a simple and effective approach for the performance evaluation of a storage scheme, and demonstrated its usage for several file structures. We proposed a new file organization, its associated maintenance strategy, and analyzed--experimentally--its performance. Much work remains to be done in this area: more detailed cost structure, more accurate estimations of the cost coefficients, extensions to linked and multi-linked files, and so forth. An additional direction for research is to extend our work to dynamic, non-stable files. As our results suggest, a partial order scheme might be advantageous for dynamic files, due to the relatively low cost associated with insertions and deletions in such schemes.

References

1. Babad, J. M. "A Record and File Partitioning Model". Research Report No. 7358, Graduate School of Business, University of Chicago, Chicago, Ill., 12/73.

2. Katter, R. V. "Design and Evaluation of Information Systems". In Annual Review of Information Science and Technology, Vol. 4, Encyclopedia Britannica, Inc., Chicago, Ill., 1969.

3. Knuth, D. E. The Art of Computer Programming. Vol. 3, Addison-Wesley Publishing Co., Reading, Mass., 1973.

4. Lancaster, F. W. and Gillespie, C. J. "Design and Evaluation of Information Systems". In Annual Review of Information Science and Technology, Vol. 5, Encyclopedia Britannica, Inc., Chicago, Ill., 1970.

5. Lefkovitz, D. File Structure for On-Line Systems. Spartan Books, New York, 1969.

6. Loeser, R. "Some Performance Tests of Quicksort and Descendants". Communication of the ACM, Vol. 17, No. 3, 3/74, pp. 143-152.

Appendix: Heap Manipulation Routine

Below we list the routine which we used for the manipulation of the heap structure. This routine inserts records into the heap, deletes records from the heap, searches for a specific record, and sorts the heap. However, this routine does not sort a file which is not structured as a heap. Many routines exist for the latter purpose, and the reader is referred to Knuth [3] or Loeser [6].

```
      SUBROUTINE HEAP(X,MX,NX,Y1,IS,MS,NOP)
C----     PARTIALLY ORDERED HEAP MANIPULATIONS ROUTINE
C----     HEAP X, MAX. SIZE MX, USED SIZE NX
C----     Y1 INPUT VALUE TO BE MANIPULATED, REDUNDANT FOR SORT
C----     STACK = IS, SIZE MS = CEIL(LOG(MX)); USED IN SEARCH
C----     NOP DESIGNATES OPERATION ON INPUT:
C----       1=SEARCH, 2=DELETE, 3=INSERT, 4=SORT
C----     ON RETURN, NOP FLAGS ROUTINE≠S SUCCESS:
C----       0=SIZE ERROR, -1=FAILED SEARCH/DELETE, ≥=INDEX FOR
C----       SUCCESSFUL SEARCH/DELETE, UNCHANGED FOR SUCCESSFUL
C----       INSERTION/SORT
C----     TEMPORARY VARIABLES NAMES ARE DERIVED FROM MAIN VARIABLE
C----       FIRST LETTER. INDEX PRECEDED BY K
      DIMENSION X(MX),IS(MS)
C----     CHECK INPUT PARAMETERS
      IF(MX.GT.0.AND.MS.GT.0.AND.NX.GE.0.AND.NX.LE.MX.AND.
     1   NOP.GE.1.AND.NOP.LE.4)GO TO 10
C----     ERROR IN INPUT OR ARRAYS SIZE OR ARRAYS FILLED
    5 NOP=0
      RETURN
C----     PARAMETERS O.K.          BRANCH TO OPERATION
   10 GO TO (70,70,220,15),NOP
C>>>>>>>>>>>>> SORT -- FINAL STAGE -- HEAP TO ASCENDING ORDER
C----     METHOD -- ROOT MOVED TO END, WHILE LAST ELEMENT
C----       BUBBLED FROM ROOT TO LEAVES
   15 KX=NX
   20 IF(KX.LE.1)RETURN
      X1=X(KX)
      X(KX)=X(1)
      KX=KX-1
      KX1=1
   30 KX2=KX1+KX1
      IF(KX2-KX)40,50,60
   40 IF(X(KX2).LT.X(KX2+1))KX2=KX2+1
   50 IF(X1.GE.X(KX2))GO TO 60
C----     BUBBLE VIA LARGEST SON
      X(KX1)=X(KX2)
      KX1=KX2
C----     TRY TO BUBBLE TO LOWER LEVEL
      GO TO 30
   60 X(KX1)=X1
      GO TO 20
C>>>>>>>>>>>>> SEARCH .... KX 0 IF NO LUCK, ELSE INDEX
   70 KX=0
      KNOP=NOP
C----     TEMPORARY FOR NOP       NOP IS SET FOR FAILURE
      NOP=-1
      IF(NX.LE.0)RETURN
      KS=0
      KX1=1
C----     KS STACK POINTER, KX1  CURRENT NODE (INITIALLY ROOT)
   80 IF(Y1-X(KX1))90,140,130
   90 KX1=KX1+KX1
C----     SEARCH SONS
      IF(KX1-NX)100,80,130
  100 KX2=KX1+1
C----     COMPARE BROTHERS         ASSUME MAX = RIGHT ONE
      IF(X(KX1).LE.X(KX2))GO TO 110
      KX1=KX2
C----     MAX = LEFT BROTHER
      KX2=KX1-1
  110 KS=KS+1
C----     MAX SON TO TOP OF STACK
```

```
      IS(KS)=KX2
      GO TO 80
  130 IF(KS.LE.0)RETURN
C----      RETURN ON EMPTY STACK: ELSE TAKE NODE ON TOP
      KX1=IS(KS)
      KS=KS-1
      GO TO 80
  140 NOP=KX1
C----      SUCCESSFUL SEARCH
      IF(KNOP.EQ.1)RETURN
C>>>>>>>>>>>>>> DELETE
      IF(NX.GT.1)GO TO 170
C----      DELETE LONE ELEMENT
      NX=0
      RETURN
  170 X1=X(NX)
C----      NOT LONE ELEMENT      INSERT LAST ONE IN PLACE OF DELETED
      NX=NX-1
      KX=NOP
      IF(KX.LE.1)GO TO 180
      KX1=KX/2
C----      COMPARE WITH FATHER
      IF(X1-X(KX1))180,210,250
C----      LESS THAN FATHER      BUBBLE AS IN SORT
  180 KX1=KX+KX
      IF(KX1-NX)190,200,210
  190 IF(X(KX1).LT.X(KX1+1))KX1=KX1+1
  200 IF(X1.GE.X(KX1))GO TO 210
      X(KX)=X(KX1)
      KX=KX1
      GO TO 180
  210 X(KX)=X1
      RETURN
C>>>>>>>>>>>>> INSERT -- BUBBLE TO ROOT
C----      ALSO USED BY DELETE
  220 IF(NX.GE.MX)GO TO 5
      X1=Y1
      NX=NX+1
      KX=NX
  240 IF(KX.LE.1)GO TO 260
      KX1=KX/2
C----      KX1 IS FATHER
      IF(X1.LE.X(KX1))GO TO 260
  250 X(KX)=X(KX1)
      KX=KX1
C----      TRY TO CONTINUE BUBBLING
      GO TO 240
  260 X(KX)=X1
      RETURN
      END
```