01 Dec 1987

Performance Parameter Measurements of Generic Files

Sankarraman Subramanian

George Winston Zobrist
Missouri University of Science and Technology

PERFORMANCE PARAMETER MEASUREMENTS OF GENERIC FILES

Sankarraman Subramanian* and George Zobrist

CSc-87-15

*This report is substantially the M.S. thesis of the first author, completed December, 1987.

ABSTRACT

This study discusses the performance parameter measurements of generic files, the pile file, the sequential file, the indexed-sequential file, the indexed file and the direct file. The file performance measurements are compiled in a software package. The study then describes the use of such software package as a simulation tool in a file design environment.

TABLE OF CONTENTS

I. INTRODUCTION

Files are generally characterized by their storage organization on external devices. Different organization leads to differences in performance while storing and retrieving records. It is well known that main storage access time is at least four to five orders of magnitude faster than secondary storage access time. Hence, the objective in any application, like a database management system [1] is to organize the data efficiently and thereby minimize the number of external accesses.

There is no single storage structure that is optimal for all applications. The process of choosing a storage structure for complex applications is nontrivial since such applications have to work within time and size constraints. In the design phase most requirements tend to conflict with each other. Also a given application assumes that services are provided by the operating system for building its file system.

The basic building block to a file system design is an analysis of generic file organizations [2]. This analysis includes not only the discussions on storage structure of these files but also its effects on performance measurement of such files and hardware analysis on which the data is stored.

In this thesis a consistent view of generic file organizations is provided and the performance equations for generic organizations as derived in Ref [2] are used. The performance parameters of the generic file organizations are compiled in a software package. The package can be used to find the performance parameters of generic files for any

storage device. The purpose of the thesis is to demonstrate the use of such a tool in a teaching and research environment. The earlier chapters provide a description of each generic file. The thesis concludes with a discussion on software organization and its application in a teaching and research environment.

II. HARDWARE PARAMETERS AND QUANTITATIVE MEASURES

The hardware parameters that will be used for measures of file system performance will be discussed here. The primary interest is in a small number of hardware parameters like seek time (s), rotational latency (r), block transfer time (btt) and bulk transfer rate (t'). Finally the seven quantitative measures that form the basis of the file performance evaluation are discussed.

A. RECORDS AND BLOCKS

A record is defined as a collection of fields about an entity of interest. It is the actual unit of information at the logical or file level. However, to reduce the gaps that are formed, records are grouped together to form a physical record or block. A block is the unit of data transferred between core memory and external storage device. Selection of optimum block size is influenced by factors like size of buffer, time required to transfer the block and the capacity of a track. The record size is denoted by R and the block size by B.

Since the file performance is in terms of records, some parameters which relate block based parameters to records are required. Records may be of fixed or variable length. The number of records that fit into a block is known as blocking factor (Bfr).

1. Fixed Blocking. If records are of fixed length, the blocking Factor (Bfr) is given by

$$Bfr = \left\lfloor \frac{B}{R} \right\rfloor \tag{2.1}$$

2. Variable Length Blocking. In order to manipulate records that are variable length it is necessary to add record marks of size P, at the beginning of each record indicating the length of the record. With variable length blocking an average of 1/2 R is wasted due to filling problem. The value of record size (R) is an average value. Hence the blocking factor (Bfr) for varying length record is

$$Bfr = \frac{B - \frac{1}{2} \times R}{R + P} \tag{2.2}$$

B. WASTE

There are gaps between blocks due to hardware design and there are unused space within blocks and various markers. The gap size (G) due to the hardware design between each block is given by the manufacturer. Hence the waste due to gaps (W_G) per record is

$$W_G = \frac{G}{Bfr} \tag{2.3}$$

There is also waste due to unused space from blocking in each block. This is allocated to each record as W_R. If the records are of fixed length the wasted space due to blocking is less than the size of one record (R). The value of W_R is bounded by $0 \leq W_R \leq \frac{R}{Bfr.}$ For fixed blocking the waste per record is given by

$$W = W_G + W_R, \text{ often } W \simeq \frac{G}{Bfr} \tag{2.4}$$

If the records are of variable length, on an average one half record per block is wasted due to accommodation problem. Also each record requires a marker entry. Hence W_R for variable length record is

$$W = P + \frac{\frac{1}{2} \times R + G}{Bfr} \qquad (2.5)$$

C. ACCESS TIME

The time required to transfer a block is the sum of seek time (s), rotational latency (r) and block transfer time (btt).

D. SEEK TIME (S)

Seek time is defined as the time required to position the head mechanism over the track that contains the block to be accessed. For the file system performance measurement a seek time provided by the manufacturer for the disk device chosen is used.

E. ROTATIONAL LATENCY (R)

Since the capacity of the track is quite large the tracks are divided into units called blocks. After the disk head is positioned at the right track there is a rotational delay incurred to reach the desired block on the track. This delay is referred to as rotational latency.

The average value of rotational latency (r) is one half the time required for one rotation of disk.

$$r = \frac{1}{2} \frac{60 \times 1000}{rpm} \quad ms \qquad (2.6)$$

where rpm is the disk revolutions per minute and rotational latency is specified in milliseconds.

F. BLOCK TRANSFER TIME (BTT)

After positioning the head over the proper track and over the beginning of the block desired, the actual data block still has to be read from or written to the disk. The rate at which data can be transferred is known as the transfer rate (t). This basic transfer rate is dependent on the device used. The rate is provided by the manufacturer. The time required to transfer a block is known as block transfer time (btt). It is given by

$$btt = \frac{B}{t} \quad ms \tag{2.7}$$

The transfer rate (t) provided by the manufacturer is an instantaneous rate of data transfer. While reading large quantities of data we have to account for time taken, when gaps and waste areas passes under the head mechanism. Also, correction has to be made for seek time that occurs at the end of each cylinder. This corrected transfer rate known as bulk transfer rate (t') [2] is

$$t' = t_{cyl} + s' \quad \text{characters/ms} \tag{2.8}$$

$$t' = \frac{R}{(R + W)/t + s'} \quad \text{characters/ms} \tag{2.9}$$

$$t' = \frac{t}{2} \frac{R}{R + W} \quad \text{characters/ms} \tag{2.10}$$

G. UPDATING BLOCKS

Updating data in a block is an expensive operation because it requires at least one read and one write operation. During an update operation the block containing the desired record is read into main memory, desired changes to the fields of the record are made and the block is written back into its original position. Inserting a record to the end of the file is similar to an update operation. The last block is read into core memory, the record is inserted into the block and the block is rewritten. In either case, if the necessary insertion or changes can be made fast enough, the block can be rewritten during the next disk revolution. Such rewrite operation will take one revolution or approximately

$$T_{rw} = 2r \tag{2.11}$$

if time to update in memory is $<< 2r$

H. QUANTITATIVE MEASURES

There are seven quantitative measures necessary for evaluating each of the five file organization methods. The seven measures are :

1. Record Size (R). This contains the amount of storage required for the record. This includes the amount of space occupied by the actual data, the pointers and record markers, if any, to represent the record and any index or pointer overhead that was built for purpose of efficient access.

2. Fetch Record. Fetching a record is a two step process. The actual location of the record and then the reading of the block into the buffer. We assume that the retrieval of a record is random and no preparation for such a fetch has been made. T_F gives the time to fetch a record.

3. Get-Next Record. Records that are related are usually fetched together. Successor records are obtained more efficiently, if locality of data is strong. This quantitative measure will give the time to fetch the next record within the file. T_N denotes time to get the next record.

4. Insert Record. Since most files are volatile, new data records need to be added for the file to remain up to date. Also, the writing operation involves more overhead than reading because the write operation is actually a read and rewrite operation. Adding records to the end of the file is easy, but if data is clustered in the file according to a field, records have to be shifted to accommodate the insertion of the new record. T_I gives the time to update a file by inserting a record.

5. Update Record. T_U gives the time to update a record by changing the data within the stored record. It takes more time to update a record where changes are made to the key field. Such an update is done by marking the record as deleted and inserting one in sequence, with the updated key field.

6. Read Entire File. In some applications, reading the entire file is often done at regular intervals. This might be done for preparing payroll at the end of a month or to produce a list of records that matches an uncommon search attribute. T_X gives the time for exhaustive reading of the entire file.

7. Reorganization of File. It is necessary to clean up files periodically for removing deleted records and rearranging inserted ones. The time between such reorganization depends both on application requirements and the type of file organization used. T_Y gives the time needed for reorganization of the file.

III. THE PILE FILE

The pile file does not possess an efficient data organization. The data records in the pile file are stored in the order of their arrival. Records may be of variable length and individual data elements between records may be completely different. The pile file may be visualized as a repository of not necessarily related information. Individual fields in a record of a pile file are actually an attribute name - value pair. Thus the data record will be of the form

name=SANKAR, number=81621, height=170;

Information is retrieved from a pile file by specifying some attributes as search argument and retrieving other attributes as goal data. A pile file is used in places where data is not easy to organize or where data is collected prior to processing.

A. RECORD SIZE

In a pile file additional overhead is encountered since we need to store the attribute name for each data field in a record. On the other hand, individual fields unrelated to the record need not be stored at all. The effect is a relatively high density when data elements collected are heterogeneous and low density when data elements collected are homogeneous. Each data element also needs two different characters to mark the end of an attribute name and a data value. Assuming the average length of description of an attribute as

'A' and data value as 'V', the expected average length of a pile record is given by

$$R = a'(A + V + 2) \tag{3.1}$$

B. FETCH RECORD

Since the records in the pile file are not grouped into any order, the time required to locate a record is high. The desired record may be the first one or the last one. The expected average then, is the sum of all the times to reach and read any of the blocks, divided by the number of choices.

$$\text{Average blocks read} = \sum_i \frac{i}{b} \tag{3.2}$$

if $b >> 1$ and $i = 1 \ldots b$

The time to read this number of blocks sequentially using Eq.2.10 is given by the number of blocks to be read times the time to read one block

$$T_F = \frac{1}{2} \times b \times \frac{B}{t'} \tag{3.3}$$

$$T_F = \frac{1}{2} \times n \times \frac{R}{t'} \qquad \text{since } nR = bB \tag{3.4}$$

The use of bulk transfer rate is appropriate here because we read the file sequentially passing over gaps and cylinder boundaries until the desired block is read.

C. GET-NEXT RECORD

Records are not kept in any order in the pile file. They are stored in the order of their arrival. The successor record may be anywhere in the file, in which case, it is similar to a fetch request. The time required to find the successor record is

$$T_N = T_F = \frac{1}{2} \times n \times \frac{R}{t'} \tag{3.5}$$

D. INSERT RECORD

Records are inserted (added) to the end of the file. If the address of the end of the file is known, a new record is added to the end of the file and the end pointer is updated. The time for insert will then be the sum of the times to read a block, the time to append the record and the time to rewrite it.

$$T_I = s + r + btt + T_{RW} \tag{3.6}$$

Assuming that Time to rewrite is equal to 2 * r from EQ.2.11

$$T_I = s + 3r + btt \tag{3.7}$$

E. UPDATE RECORD

Since the pile file may consist of records that are unrelated and records may not have similar sets of elements, updating a record is done by fetching the old record, marking the record as invalid and inserting a new one (probably larger) at the end of the file.

$$T_U = T_F + T_{RW} + T_I \tag{3.8}$$

$$T_U = \frac{1}{2} \times n \times \frac{R}{t'} + s + 5r + btt \tag{3.9}$$

In the case of deletion of a record the T_I term drops off.

F. READ ENTIRE FILE

An exhaustive search of the entire file is twice as costly as a single fetch, if the order of retrieval of the records does not matter

$$T_X = 2 \times T_F = n \times \frac{R}{t'} \quad \text{<sequential>} \tag{3.10}$$

However, if the file has to be read serially according to some attribute, the cost is n times an individual fetch. Hence, $T_X = n \times T_F$. Since the cost is extremely high, an alternate method is to sort the file according to the search attribute and then exhaustively read the sorted file.

Sorting : There are external routines which can be performed in $O(n \log_2 n)$ steps. The $\log_2 n$ term refers to the number of passes over the file after initially sorting the values in each block. In subsequent passes the sorted blocks are merged into a sorted sequence of blocks until the entire file is merged. Each sort step involves a block read and a block write operation. The time required to perform the sort over the entire file is the product of the time to sort the records within a block and the time to merge the sorted blocks into sorted sequence

$$T_{sort}(n) = 2 \times b \times btt + 2 \times b \times \lceil \log_2 b \rceil btt \quad (3.11)$$

$$T_{sort}(n) = 2 \times n \times \left\lceil 1 + \log_2\left(\frac{n}{Bfr}\right) \right\rceil \frac{R}{t'} \quad (3.12)$$

The time to exhaustively read the file serially is

$$T_X = T_{Sort}(n) + T_X Seq \quad (3.13)$$

G. REORGANIZATION OF PILE

Reorganization of the pile file is done periodically by removing the records that are marked as deleted or changed and reblocking the remaining records into a new file. If the number of records added between reorganization is 'o' and number of deletion is 'd', the time to reorganize the file is

$$T_Y = (n + o)\frac{R}{t'} + (n + o - d)\frac{R}{t'} \quad (3.14)$$

Here o equals n(insert) + v and d equals n(delete) + v where n(insert) is the number of records inserted, n(delete) is the number of records deleted and v is the number of records updated.

IV. THE SEQUENTIAL FILE

Records in the sequential file are ordered into a sequence known as key sequence. The key for each record is defined as one or more fields within the record that uniquely distinguishes the record from every other record in the file. The records in a sequential file are of fixed length and each record contains the same number of attributes.

One of the disadvantages of this file organization involves updating of records in the file. Records have to be moved in the main file, to accommodate insertion of a new one to maintain the key sequence. Also, changes to the key field of a record disturbs the sequence.

Insertion of a new record into the sequential file is handled by placing the records in the order of arrival in a separate transaction log file. At the time of reorganization a batch update is performed by reading records from both the main and log file. Records are merged into a new file in key sequence.

A. RECORD SIZE

Unlike pile file, attribute name need not precede data value because all the records in the sequential file contain the same number of attributes occupying the same position within the record. If 'a' is the number of attributes and 'V' is the average length of the data values, the record size is given by

$$R = a \times V \tag{4.1}$$

B. FETCH RECORD

There are two methods for fetching a record from a sequential file. If the search argument is not the key attribute, at least half of the main file and the overflow file has to be searched. If 'o' is the number of records that can be accommodated in the overflow file then the time to fetch a record is given by

$$T_{F1} = \frac{1}{2} (n + o) \times \frac{R}{t'} \tag{4.2}$$

If search argument is the key attribute then a binary search technique is adopted to access the data desired in the main file. In the binary search technique, during each pass the search space is dissected by half until the desired record is found. Also the overflow file has to be searched. Since new records are appended to the overflow file, on an average half of it is searched to locate the desired record. In this case

$$T_{F2} = \log_2(\frac{n}{Bfr}) \times (s + r + btt) + \frac{1}{2} \times o \times \frac{R}{t'} \tag{4.3}$$

C. GET-NEXT RECORD

Since records are kept in sequence the next record is already available in the buffer. For every Bfr records the next block must be transferred into core memory to obtain the next record.

$$T_N = \frac{btt}{Bfr} \tag{4.4}$$

D. INSERT RECORD

Insertion of new records into the sequential file is done by appending the records to the overflow file and periodically reorganizing the main file and the overflow file into a single sequential file. The cost incurred in reorganization should be included in the cost of insertion.

$$T_I = s + r + btt + T_{RW} + \frac{T_Y}{o} \tag{4.5}$$

$$T_I = s + 3r + btt + \frac{T_Y}{o} \quad \text{<using Eq.2.11>} \tag{4.6}$$

E. RECORD UPDATE

The time to update a record that involves no changes to the key field is sum of the time to fetch the record and the time to rewrite it. However, if changes to the key field are necessary, the update is done by inserting two records to the overflow file. One is the updated record and the other a flag record indicating the deletion of the old one.

$$T_U = T_F + T_I \tag{4.7}$$

$$T_U = \log_2(\frac{n}{Bfr}) \times (s + r + btt) + \frac{1}{2} \times o \times \frac{R}{t'} + T_I \tag{4.8}$$

F. READ ENTIRE FILE

Reading of the entire file is done by sorting the overflow file into key sequence and then reading the main file and the overflow file sequentially.

$$T_X = T_{sort}(o) + (n + o) \times \frac{R}{t'} \tag{4.9}$$

where T_{sort} is given by Eq.3.12.

G. REORGANIZATION OF SEQUENTIAL

This involves sorting the overflow file and then merging both the main file and the sorted file into a new sequential file. Merging requires reading both the main and the overflow file and writing to a new file.

$$T_Y = T_{sort}(o) + (n + o) \times \frac{R}{t'} + n_{new} \times \frac{R}{t'} \tag{4.10}$$

where n(new) = n(old) + n(insert) - d and T(sort) is given by Eq.3.12.

V. THE INDEXED SEQUENTIAL FILE

Indexed sequential file is an improvement on the sequential file organization. It provides better random access to individual records by means of an index to the primary file. Also insertion of new records are handled differently than in a sequential file organization. When records are inserted, pointer adjustments are made to allow efficient serial reading of the file. The indexed sequential file has three major components: the sequential file, the index and the overflow area. The structure of the data file is similar to the sequential file seen in a previous chapter.

A. INDEX

The important benefit of building an index to the main file is efficient access of individual records. An index is a set of entries each containing two fields. One is the key attribute of the record and the other a pointer to the location of the record. Indexes are kept in a serial order according to the key attribute. The first level index pointers will contain the address of data records. Additional indexes are built to index the previous levels until the top level master index occupies one block. This block is brought into core memory when the file is opened.

By using block anchors each index entry can refer to a block of records in the main file, instead of a single record. Individual records are found by searching within the block. This would reduce the number of index entries to n/Bfr. An important parameter of an index is its fanout ratio. It is defined as the number of blocks (in a block

anchored index), a block of index entries can reference. It is given by

$$y = \left\lfloor \frac{B}{V + P} \right\rfloor \tag{5.1}$$

In a hardware oriented index design there are usually two level of indexes. The first one will reference individual cylinders and the second one references the blocks inside that cylinder. The number of index level required is

$$x = \left\lceil \log_y \left\lceil \frac{n}{Bfr} \right\rceil \right\rceil \tag{5.2}$$

B. OVERFLOW

Insertion in the indexed sequential file is handled by using a technique called push through. This provides efficient serial access to the entire file according to the key attribute. Extra space is provided in each cylinder to insert new records. Addition of new records will require only rotational latency but not a seek overhead.

Push through : Each block of records contain an overflow pointer. Records are maintained in key sequence in the primary file. Insertion of a new record might lead to moving of records within the block. The record at the end of the block gets pushed to the overflow area. Each record in the overflow area contains a pointer. The pushed through record is placed in the overflow area in the next space available but pointer adjustments are made to maintain serial access to the record.

C. RECORD SIZE

Each record requires space for 'a' data values and a pointer entry 'P'. In the main file the pointer in individual records can be used as a marker for deleted records. The record size is

$$R = aV + P \tag{5.3}$$

Also space occupied by the index must be accounted for in each record. The number of blocks occupied by the first level index is

$$B_1 = \left\lceil \frac{\frac{n}{Bfr}}{y} \right\rceil \tag{5.4}$$

Subsequent levels are obtained by dividing previous levels by the fanout ratio until the value of b_i becomes one. The space for index is then

$$SI = (bi_1 + bi_2 + \ldots\ldots + 1)B \tag{5.5}$$

The space for each record is

$$R_{total} = R + \frac{o}{n}R + \frac{SI}{n} \tag{5.6}$$

D. FETCH RECORD

The index is used to locate the records. The record will be found in either 1. the main file, or 2. directly in the overflow file as the first record, or 3. indirectly in the overflow file by following a chain.

1. T fetch main. The main index is already in core memory and hence (x -1) index levels must be accessed. If all index levels are placed in the same cylinder as the data, accessing index levels requires only rotational latency and the time for reading a block. The time required for locating a record in the main file is

$$T_F main = s + (x - 1)(r + btt) + r + btt \quad (5.7)$$

2. T fetch overflow. If o' is the number of records the file has received since the file was reorganized with 'n' records, the probability of overflow per record is

$$P_{ov} = \frac{o'}{n + o'} \quad o' \text{ usually taken as } 0.5o. \quad (5.8)$$

Fetching a record that is placed first in the chain in the overflow area is

$$T_F overflow = P_{ov}(r + btt) \quad (5.9)$$

Space for overflow is provided in the same cylinder.

3. T fetch chain. Records that are pushed from the primary block to the overflow area are placed in a chain to the primary block. The length of each chain is

$$L_C = P_{ov} \times Bfr \quad \text{if } Bfr >> 1 \quad (5.10)$$

Assuming that $(L_C - 1)$ records are placed in different blocks the number of additional block access required to locate the desired record is $((L_C - 1) + 1)/2$. Therefore

$$T_F chain = P_{ov} \frac{L_c}{2}(r + btt) \tag{5.11}$$

Time to fetch a record is

$$T_F = T_F main + T_{overflow} + T_F chain \tag{5.12}$$

$$T_F = s + (x + \frac{o'}{n + o'} + \frac{Bfr}{2}(\frac{o'}{n + o'})^2)(r + btt) \tag{5.13}$$

E. GET-NEXT RECORD

To locate the next record there are number of possibilities depending on the location of the predecessor and successor records. There are several cases and evaluation of different cases is based on the probability of occurrence of the event. For a detailed analysis the following notations are useful.

P_d: the current record is in primary data block = $1 - P_{ov}$

P_b: the successor record is in same data block = $1 - 1/Bfr$

P_m: there is no insertion into the block = $1 - P_{ov}$

P_c: the next block is in the same cylinder = $1 - 1/\beta$

where β is number of cylinders per block.

P_l: the current overflow record is not the last in chain = $1 - 1/L_c$

1. case 1. Current record is in the main file and successor record is in the same block.

$(P_d) \times (P_b)$

$(1 - P_{ov}) \times (1 - 1/Bfr)$

2. case 2. Current record is the last one in the block, there are no insertion to the block and next record is the first one in next block.

$$(P_d) \times (1 - P_b) \times (P_m) \times (P_c) \times (r + btt)$$

$$(1 - P_{ov}) \times (1/Bfr) \times (1 - P_{ov}) \times (1 - 1/\beta) \times (r + btt)$$

3. case 3. Current record is the last one in the block, there is no insertions to the block and next record is in a new block on another cylinder.

$$(P_d) \times (1 - P_b) \times (P_m) \times (1 - P_c) \times (s + r + btt)$$

$$(1 - P_{ov}) \times (1/Bfr) \times (1 - P_{ov}) \times (1/\beta) \times (s + r + btt)$$

4. case 4. Current record is the last one in the block, there is an insertion and successor record is in the overflow block.

$$(P_d) \times (1 - P_b) \times (1 - P_m) \times (r + btt)$$

$$(1 - P_{ov}) \times (1/Bfr) \times (P_{ov}) \times (1/\beta) \times (s + r + btt)$$

5. case 5. Current record is an inserted record and successor record is on another overflow block and obtained by following the chain.

$$(1 - P_d) \times (P_1) \times (r + btt)$$

$$(P_{ov}) \times (1 - 1/L_c) \times (r + btt)$$

6. case 6. Current record is an inserted record in the overflow area but last in the chain and next record is obtained from the next block in main file.

$$(1 - P_d) \times (1 - P_1) \times (r + btt)$$

$$(P_{ov}) \times (1/L_c) \times (r + btt)$$

If cylinder seek is ignored and overflow area is in the same cylinder, the time to get next record is

$$T_N = \frac{n + o'Bfr}{(n + o')Bfr}(r + btt) \tag{5.14}$$

F. INSERT RECORD

When a new record is added, the overflow area of the cylinder needs to be accessed. Either the new record or the last record in the primary block gets pushed to the overflow area. Pointer adjustments are made to maintain serial access. Accessing the overflow block in the same cylinder would require a rotational latency and a block read only.

The time to insert a record would then require time to fetch the predecessor record, time to rewrite the block with the inserted record, time to access the overflow block and time to rewrite that block with the pushed through record.

$$T_I = T_F + T_{rw} + r + btt + T_{rw} \tag{5.15}$$

$$T_I = T_F + 5r + btt \quad \text{<using Eq.2.11>} \tag{5.16}$$

G. UPDATE RECORD

Analysis of a general case that allows changes to any field in the record would require fetching the record, rewriting the record with a marker to indicate deletion and inserting the record with the field updated.

$$T_U = T_F + T_{rw} + T_I \tag{5.17}$$

$$T_U = 2 \times T_F + 7r + btt \tag{5.18}$$

H. READ ENTIRE FILE

Serial reading of the entire file according to the key attribute is is easier in this file organization since seriality is maintained even between records in the overflow area by use of pointers. The index can be ignored. The time to read the entire file is

$$T_X = T_F + (n + o' - 1)T_n \tag{5.19}$$

$$T_X \simeq \frac{n + o'Bfr}{Bfr} \times (r + btt) \quad \text{<serial>} \tag{5.20}$$

Reading the file sequentially would require

$$T_X = (n + o')\frac{R}{t'} \quad \text{<sequential>} \tag{5.21}$$

I. REORGANIZATION OF FILE

Reorganization of the entire file requires reading the primary and overflow file and rewriting them into a single new file after omitting the records that were deleted. Assuming records chained to the overflow area are on different blocks, accessing records in the chain would require a rotational latency and a block access. Also a new index is built after reorganization.

$$T_Y = \frac{n + o'Bfr}{Bfr}(r + btt) + (n + o' - d)\frac{R}{t'} + \frac{SI}{t'} \qquad \text{<o prime = 0.8 o>} \qquad (5.22)$$

VI. THE INDEXED FILE

In an indexed file records are not ordered by the key attribute. The placement of a record is influenced by factors such as ease of data management or reliability. Access to individual records is only through one or more indexes. If an index is available for all the attributes in the file, the file is called a fully inverted file. The records can be placed anywhere in the file as long as a pointer exists in some index to fetch the record. The record can be of varying length.

An index for an indexed file consists of a set of entries, one for each record in the file. Block anchors cannot be used in this organization since records are not sequential. The entries themselves are ordered by attribute values, each entry consisting of an attribute name and a pointer to the location of the record. A successor record in an indexed file is accessed only through use of an index.

The index can be exhaustive or selective. Exhaustive indexes have pointers for every record in the file. Selective ones have pointers only for records whose values are significant. Usually a file has at least one exhaustive index to enable processing of all the records in the file.

A major problem with the indexed organization is the need to update indexes when records are inserted, deleted or changed. It would require insertion, deletion, or both, of an index entry in the appropriate block. In order to allow dynamic changes to the index structure a structure known as a B-tree [3] is used.

A. B-TREES

In a B-tree each index block contains a set of entries. The entries consist of an attribute name and a pointer value. At the level one index all pointers determine the location of the data record. Indexes are built to index previous levels. In this case the index pointers points to the address of lower level blocks. The B-tree index blocks are at least half full. The effective fanout varies from y/2 and y.

When a record is inserted, an entry is made in the appropriate index block in the level one index. If the block is full, a new block is acquired and half the entries are moved from the full block to the new one. This is termed as block split. An entry for the record inserted is made to the appropriate block. Also, a new entry is made at level two to point to the newly acquired block. This new entry is the former $v_{y/2 + 1}$ taken from the split block. If the block at level two is full the sequence may propagate to the root block which itself would split into two leading to a new root block.

Deletion of records would lead to removal of entries. If the density of the index is less than y/2, the adjacent block is inspected. If sum of the number of entries in both index blocks is less than y, the blocks are combined into a single block. Similarly this sequence might lead to the removal of the root block itself.

Given there are n' records with indexable attributes then the number of levels is given by

$$x = \lceil \log_{yeff} n' \rceil \qquad (6.1)$$

B. RECORD SIZE

The space required for each record in the main file is

$$R_{main} = a'(A + V + 2) \tag{6.2}$$

Let 'a' be the total number of attributes in the file and a' be the average number of attributes per record. The average number of index entries referring to data records is

$$n' = n\frac{a'}{a} \tag{6.3}$$

The size of each index is $V_{index} + P$. The space occupied by each indexes for level one is

$$SI_1 = n'\frac{(V_{index} + P)}{dens} \tag{6.4}$$

The total space for all indexes for all levels is

$$SI_{total} = a \sum_{1}^{x} SI_i \tag{6.5}$$

$$R_{total} = R_{main} + a' R_{index} \tag{6.6}$$

C. FETCH RECORD

A record is fetched only through the use of index. Each index level is assumed to be on a separate cylinder. The index and data occupy different cylinders. The time to fetch a record is then

$$T_F = x(s + r + btt) + s + r + btt = (x + 1)(s + r + btt) \tag{6.7}$$

where x is given by Eq.6.1

D. GET-NEXT RECORD

The index block last used is kept in core memory and hence time to fetch the next record would incur only a block fetch.

$$T_N = s + r + btt \quad \text{if } y_{eff} >> 1. \tag{6.8}$$

E. INSERT RECORD

Records are inserted in any free area. The time to insert into the main file is referred to as T_{data}. After the record is inserted, all a' indexes referring to existing attributes of this record must be updated. This would require searching of all the index levels for each of the a' indexes from the root. Appropriate entry is made in level one block to point to the inserted record. This is referred to as T_{index}. The probability Ps that this block is split is $1/(y/2) = 2/y$. If the block is split, a new block is fetched and entries are distributed between the split block and new one. This would require rewriting of split and new blocks. This is referred to as T_{split}. The time required to insert a record is then

$$T_I = T_{data} + a'(T_{index} + P_s T_{split}) \tag{6.9}$$

$$T_I = s + r + btt + T_{rw} + a'(x(s + r + btt) + T_{rw} + \frac{2}{y}(s + r + btt + 2T_{rw})) \tag{6.10}$$

F. UPDATE RECORD

An update changes one data record and a_{update} indexes. A record is updated in a indexed file by marking it as deleted and inserting the record in a new place. This would require fetching the record, rewriting it with a marker as deleted and writing a new copy with necessary changes. If the field updated is far removed from the previous one, an entry into the level one index would occur on a different block. This requires searching for and rewriting two index blocks. The old entry is removed from one block and a new entry is inserted into another. Also, inserting or deleting entries from a block might lead to split or join of blocks.

Since the record is inserted in a new area ($a' - a_{update}$) indexes must be fixed. The pointers in these indexes must be changed to reflect the new location. Therefore time to update a record is

$$\begin{aligned} T_U = {} & T_F + T_{rw} + T_{newcopy} + \\ & 2 \times a_{update}(T_{index} + P_s T_{split}) + \\ & (a' - a_{update})(T_{fixpointer}) \end{aligned} \tag{6.11}$$

$$\begin{aligned} T_U = {} & (x + 1)(s + r + btt) + T_{rw} + s + r + btt + T_{rw} \\ & + 2a_{update}(x(s + r + btt) + T_{rw} + \frac{2}{y}(s + r + btt + T_{rw})) \\ & + (a' - a_{update})(x(s + r + btt) + T_{rw}) \end{aligned} \tag{6.12}$$

G. READ ENTIRE FILE

The indexed file is unsuitable for exhaustive searches. When required, an exhaustive index is used to search the entire file. A brute force approach using such index would cost

$$T_X = T_F + (n-1)T_n \tag{6.13}$$

H. REORGANIZATION OF FILE

Indexed files are not as dependent on reorganization as other files. Reorganization is done to remove any poorly distributed indexes. If the data file is also reorganized it would require reading the old file, writing a new one and reconstructing the indexes. Reconstruction of an index is done by reading the file and collecting all the attributes then sorting them and building an index from the sorted file. Time to reconstruct one index is

$$T_{Yi} = T_X + T_{sort}(n') + \frac{SI}{t'} \quad \text{<one index>} \tag{6.14}$$

Time required to reorganize the data and all indexes is

$$T_Y = 2 \times T_X + aT_{Yi} \quad \text{< data and indexes>} \tag{6.15}$$

VII. THE DIRECT FILE

The direct file is not an extension of the file organizations seen earlier. In this organization the records are located by means of an address obtained by performing a computation on the key attribute of the record. The address thus obtained is a relative address within the file. Access to the records is only through this single key attribute. Records related to each other by key sequence hardly appear in physical sequence. The space occupied by a record is referred to as a slot in the direct file. Generally slots are grouped together to form a bucket. The size of a bucket equals the size of a block. In general, extra slots are provided while allocating file space to accommodate insertion.

The procedure adopted to compute the address may be termed deterministic, which generates unique addresses, or non-deterministic, which generates mostly unique addresses but does not guarantee it. Deterministic algorithms are difficult to construct and are not stable with record insertions. On the other hand, when adopting non-deterministic procedures different keys may sometimes generate the same address. This is referred to as collision. Additional cost is incurred when adopting non-deterministic procedures. The cost can be minimized by grouping slots into buckets in the main file. The bucket is searched first to identify an empty slot. If the bucket is full, collision resolution is adopted to place the record.

A. COLLISION RESOLUTION

The two common strategies for resolving collisions are linear search and separate overflow.

1. Linear Search. In this method successive blocks are searched to find an empty slot. Since extra space is provided in the file an empty slot will be found eventually. This method avoids additional seek time but tends to cluster overflows.

2. Separate Overflow. In this method records that cause bucket overflow are placed in a separate overflow file, with linkage from the primary bucket. This would require an additional seek to locate the overflow file but does not cluster overflows.

In the file analysis 'm' slots are provided for 'n' data records ($m > n$). The number of slots per bucket varies depending on the record size and block size. The overflow file is assumed to accommodate 'o' records. A pointer is provided for each record to allow linkage to the overflow file. The same entry can be used as a marker to indicate a deleted record.

B. RECORD SIZE

If 'a' is the average number of attributes in the file and 'V' is the average attribute size, the record size is given by

$$R = \frac{(m + o)}{n}(aV + P) \quad \text{<separate overflow>} \tag{7.1}$$

$$R = \frac{m}{n}(aV + P) \quad \text{<open search>} \tag{7.2}$$

C. RECORD FETCH

The time required to fetch a record is sum of a seek, latency and a block fetch. If there is a collision, another block fetch is required. The value is

$$T_F = s + r + btt + p(s + r + btt) \tag{7.3}$$

where p is the probability of collision. The value of 'p' for a single slot bucket is given by

$$p = \frac{1}{2}\frac{n}{m} \quad \text{< separate overflow>}$$

$$p = \frac{1}{2}\frac{n}{m-n} \quad \text{< linear search >}$$

The value of 'p' for multi-slot buckets is fully discussed in Ref [4].

D. GET-NEXT RECORD

Since the records are placed randomly from the address obtained from the key transformation algorithm, the get-next record is similar to fetch record.

$$T_N = T_F \tag{7.4}$$

$$T_N = s + r + btt + p(s + r + btt) \tag{7.5}$$

E. INSERT RECORD

When records are inserted, the primary bucket is fetched first. The bucket is searched to locate an empty slot. If an empty slot is found the record is inserted in the bucket. If not additional fetches are required to find an empty slot. The probability that the initial slot is filled, for a single slot bucket is given by

$$pl_u = 1 - e^{n/m} \quad < \text{separate overflow} >$$

$$pl_u = \frac{n}{m} \quad < \text{linear search} >$$

For multiple slots per bucket the value of pl_u is given in Ref [4]. The cost of insertion is the sum of the cost of finding a empty slot in the initial fetch and the cost when it is full.

$$T_I = s + r + btt + T_{rw} + (1 - pl_u)\ (s + r + btt) \tag{7.6}$$

F. UPDATE RECORD

The time to update a record that does not change the key field is sum of fetching a record and rewriting it with the updated fields. The value is

$$T_U = T_F + T_{rw}$$

$$T_U = s + 3r + btt + p(s + r + btt) \tag{7.7}$$

G. READ ENTIRE FILE

The direct file is not well suited for exhaustive reading. When required the entire file address space is read including the overflow file. The time to read exhaustively is

$$T_X = (m + o) \frac{R + W}{t'} \tag{7.8}$$

In case of linear search the term 'o', the size of the overflow file, drops off.

H. REORGANIZATION OF FILE

Reorganization of a direct file is necessary if the number of additions to the file has increased to the point where the density n/m has exceeded design goals. More space 'm' has to be provided for the reorganized file and the key-transformation algorithm needs to be rewritten. Reorganization requires reading the file exhaustively and loading the file back into new slots.

$$T_Y = T_X + T_{load} \tag{7.9}$$

Time for loading : The loading cost can be reduced by sorting the file to be loaded. The key attribute of each record is transformed to unique addresses by the key transformation algorithm. This address is appended to each record. The records are then sorted by this address. Sorted records are then loaded into the new file after

removing the address part which was appended earlier. Time to load is

$$T_{load} = T_{sort} + T_X \tag{7.10}$$

where T_{sort} is given by Eq.3.12 and T_x by Eq.7.8

VIII. SOFTWARE ORGANIZATION

A package, GENERIC, was developed to calculate the seven performance parameters of the five file organizations. The package was compiled and stored in the IBM 4341 system library at UMR. Anyone with relevant EXEC files and permission can access the package from a CMS environment. The procedure to access the package is explained in file performance parameter manual available from computer science department and attached as an Appendix B.

The language used for coding the program was PL/1. The reason for choosing PL/1 was because the language provides different formats for input and output, several builtin routines to calculate mathematical functions and ease of error checking and control capabilities.

A. SOFTWARE MODULES

The main module displays the names of the five file organization and passes the control to the file module that was called. Three modules are associated with each of the five file organization. They are the file input module, the file parameter calculation module and the file result display module. Several other routines common to all file modules, are used to calculate the blocking factor, block transfer rate and the bulk transfer rate.

Values like the seek time, latency, transfer rate and gap size are dependent on the disk device. The values for each different disk device is kept separately in a file called DISK PARM. The user is prompted to choose a disk device. The parameters for the device chosen

are read from the file. New devices can be tested by adding the required values to the file.

B. ERROR CHECKING

For the program to behave as desired it is important that the user provide correct and meaningful values. However, the package has several error checking statements to check the input values and inform the user of any illegal ones. It checks for type mismatch, value out of range and run-time overflow errors. Also the program checks for errors like a given record size exceeding the given block size. Most of the errors are handled using PL/1 ON statements.

C. BATCH ACCESS

Additional work is needed to access the package in batch mode [Appendix B]. The package expects the values to be placed in a file, one value per line, starting in the first column. Since the nature and the number of the input value may vary depending on the file organization, the following procedure was adopted. The user will be provided with five input file templates. The file name of each of them is called INPUT and the filetype is one of PILE, SEQ, INDSEQ, IND and DIR. Input values must be entered in these files starting at the first column. A description is provided on the right side of the line to indicate what value is required on the left side. The package can be accessed through a NETSUB command from CMS. The results are sent back to the user reader files.

D. INTERACTIVE ACCESS

In this mode the user is prompted to choose a file organization. Then the user is asked to enter pertinent input values for the file organization chosen. If an error occurs, the program displays an error message and stops. Otherwise, the parameters are calculated and the results are displayed along with the given input values.

IX. APPLICATIONS

This chapter discusses the effective use of the software package as an application tool in a teaching and a research environment. It should be borne in mind that the output times provided by the package for a file design are average values. Also the package assumes that the given file design under consideration conforms to the view provided in earlier chapters. If the file organization design varies slightly from the one provided, the package may require changes such as addition or deletion of seek or latency times to tailor it to the environment. A major advantage of the package is it provides an environment for testing various file designs without going through the laborious process of physically building the design. Also the input parameters for the file can be changed and the behavior of the file can be studied. In this way it acts as a simulation tool.

A. RESEARCH ENVIRONMENT

A large information systems stores its data as files. The information system may be a simple single file system or a complex database management system [4] involving multiple files. The choice of a file organization depends on the kinds of retrieval to be performed. Each file system application has diverse requirements which must be met. Some examples are fast retrieval, efficient updates, and time to reorganize the file. Sometimes a given design may be adopted to conform to general requirements as compared to exceptions, like compromising the time to update for time to fetch a record. After noting the application systems requirements and constraints, the package can be used as a tool to choose a file design.

The first step is to use the package to find the performance parameters of a file design. Then the values produced can be checked to see if they are within the specified requirements. The package can then be used as a simulation tool to find the file design behavior under different conditions like an increase in record size, block size and file size. The decision to increase the block size may depend on the size of the buffer available. The same procedure can be adopted for other file organizations. The different values from file designs would help in weighing the different costs and to choose the optimal file design for a given application.

B. TEACHING ENVIRONMENT

Most information system and database design courses deal with physical file design. The class is exposed to generic file designs. However, it is difficult for the students to have an adequate feel for the issues involved with theoretical exposure alone. The package can be used as a tool by the students to gain a practical exposure to the file design problem. With the help of the package, the user can understand a file design problem better. The package would show the user how a file design behaves under different conditions. In a database design project, with the help of the package, the user can provide a detailed report of the database physical file design analysis.

X. CONCLUSION

A framework of five generic file organizations was provided. A package to measure the performance parameters of the file organizations for different storage devices was developed. The use of such packages as a simulation tool in a research and teaching environment was described. A listing of the program is provided in Appendix C and a test run is provided in Appendix D.

The graphics support to the package is the next development to be considered. Graphical interpretation of a file design, or data, is much easier to understand than raw data itself. The support could include showing in graphics the storage structure of the main file, overflow file, chaining of records, building of indexes, and propagation of an index split. Also provision can be made to generate graphs from the output values provided by the package. These graphs could show the effect of block size on fetch time for a file organization. The graph can be overlaid with the same effect for a different file organization.

The design can be extended to include multi-list file and can be tested with a practical file design implementation like VSAM.

BIBLIOGRAPHY

1. Date, C.J., An Introduction to Database Systems, Fourth Edition, Addision and Wesley, 1982.

2. Widerhold, Gio, Database Design, McGraw-Hill, 1983.

3. Comer, D., "The Ubiquitous B-tree", ACM Computing Survey 11, No. 2, June 1979, pp.440-445.

4. Knuth, D.E., The art of Computer Programming, Vol 3: Sorting and Searching, Addision and Wesley, 1973.

5. Tremblay, J.P. and Sorenson, P.G., An Introduction to Data Structures With Applications, McGraw-Hill, 1976.

6. Martin, J., Computer Data-Base Organization, Prentice-Hall, 1975.

VITA

Sankarraman Subramanian was born on April 4,1960 in Madras, India. He received his primary and secondary school education in Don Bosco Matriculation school, Madras, India. In July 1978, he entered Annamalai University, Chidambaram and received his Bachelor of Engineering degree in Mechanical Engineering in July, 1983.

He has been enrolled in the Graduate school of the University of Missouri-Rolla since August 1984 in the Department of Computer Science. During this period of time he has held the position of graduate research assistant in the Department of Computer Science and Department of Electrical Engineering.

APPENDIX A

NOMENCLATURE

A	average length of attribute name.
a	total number of attributes in a file.
a'	average number of attributes in a record.
B	block size.
b	number of blocks.
btt	block transfer time.
Bfr	blocking factor.
G	gap size.
m	number of slots available in the file.
n	number of records in the file.
o	number of records that can be accomodated in the overflow file.
P	space for pointer.
p	probability of collision.
R	space required to represent the record.
r	rotational latency time.
SI	space provided for index.
s	average seek time for the device.
T_F	time to fetch a record.
T_I	time to insert a record.
T_N	time to get the next record.
T_{sort}	time to sort the file.
T_U	time to update a record.

T_X	time to read the file.
T_Y	time to reorganize the file.
t'	bulk transfer rate.
V	average length of the attribute.
W	waste due to gaps per record.
x	number of levels in an index-structure.
y	fanout ratio.

APPENDIX B

FILE PARAMETER MEASUREMENT USAGE NOTES

INTRODUCTION

This document is intended for the users who would like to find out the performance parameters of the five generic file organizations the pile file, the sequential file, the indexed-sequential file, the indexed file and the direct file by accessing the package GENERIC from their CMS account. The package can be run in batch only. Seven quantitative measures are provided for each of the files. They are:

R the amount of storage required for a record.
T_F the time needed to fetch an arbitrary record from the file.
T_N the time to get the next record within the file.
T_I the time to update the file by inserting a record.
T_U the time to update the file by changing a record.
T_X the time needed for exhaustive reading of the entire file.
T_Y the time needed for reorganization of the file.

INPUT

Six files will be sent to your VMA student account. Five of them by the name INPUT PILE, INPUT SEQ, INPUT INDSEQ, INPUT IND and INPUT DIR. Receive these files into your user mini-disk. You are required to enter values in these input files. Choose the appropriate files for entering data. For example if you want to find performance parameter of pile file xedit INPUT PILE for entering data. All input files contain a set of values starting on column 1 on the left and description on the right indicating what these value stands for and also the range of values accepted. For entering data to any of the files just change values on the left side beginning in column 1. Do not change the description on the right or the order in which values appear. Meaningful data must be given. For example while entering block size care must be taken to enter values that are within the minimum and maximum range allowed for the device chosen and must also be greater than the record size. The sixth file is named GENERIC EXEC. Receive the file in your user disk.

The contents of the GENERIC EXEC file is shown below;

```
/* THIS IS AN EXAMPLE OF HOW TO RUN THE GENERIC PROGRAM */
'CP LINK C2816CSC 191 305 RR RETRY'
'ACC 305 T'
/* CHANGE FILETYPE AFTER INPUT IN THE NEXT LINE. FOLLOW IT WITH A' */
'EXEC GENERIC2 INPUT PILE A'
'REL 305'
'CP DET 305'
```

OUTPUT

The output is send to your rdr. you can PEEK and NETPRT the rdr file.

ACCESSING THE PACKAGE

Every time before submitting the job through NETSUB from CMS you have to make one change in the GENERIC EXEC file. The ft in the line
'EXEC GENERIC2 INPUT ft A'
is changed to one of these PILE, SEQ, INDSEQ, IND or DIR. Depending on the ft the appropriate input values will read from the file. Now to access the package type in from CMS

NETSUB GENERIC EXEC (SYS(EXECUTE) FILES(INPUT ft) RES(PLI)

where the filetype ft - may be PILE,SEQ,INDSEQ,IND or DIR. The ft should match the one in GENERIC EXEC file.

In the following pages you will find input data required for each file organization . We have followed that by defining a problem for that file and have shown the results obtained by using the package.

THE PILE FILE

The following data are required for pile file model performance parameter measurements. You have to enter the data in the file INPUT PILE .

```
1               file type    (1 .. 5) 1 for pile file
1024            block size (bytes) (1..99999)  (B)
100000          # of records in the main file (1..9999999)  (n)
4               pointer size (1..99) (P)
```

```
10              avg # of attributes per record (1 ..99)     (a')
20              attribute_name length (bytes) (1..300)      (A)
25              attribute value length (bytes) (1..300)     (V)
10000           # of inserts between reorganization (1..99999)
5000            # of deletes between reorganization (1..99999)
3000            # of updates between reorganization (1..99999)
1               type of device (1..4)
                      1 for IBM 3380 Disk Pack
                      2 for IBM 2319 Disk Pack
                      3 for IBM 851 8" Floppy Disk
                      4 for IBM 9 Track Magnetic Tape
```

Example;

Consider a pile file with following data

a) Block Size = 1024 bytes.
b) Number of records = 100,000 varying length
c) Average number of attributes/record (a') = 10.
d) Average attribute name length = 20 bytes.
e) Average attribute value length = 25 bytes.
f) Average number of inserts between reorganization = 10,000 (rec)
g) Average number of deletes between reorganization = 5,000 (rec)
h) Average number of updates between reorganization = 3,000 (rec)
g) use IBM 3380 Disk pack

Calculate the seven measure of performance $T_F, T_N, T_I, T_U, T_X, T_Y$

The program prints out the input data entered and then the seven measure of performance. The output should look like this;

PILE FILE INPUT DATA

Block Size;	1,024	bytes
Total number of records in the main file;	100,000	records
Average # of attributes per record;	10	attr.
Average attribute_name length;	20	bytes
Average attribute_value Length;	25	bytes
Average # of inserts between reorganization;	10,000	records
Average # of deletes between reorganization;	5,000	records
Average # of updates between reorganization;	3,000	records

PILE FILE PERFORMANCE PARAMETERS

Blocking factor;	2.00	records
Block transfer rate (btt);	0.34	ms
Bulk transfer rate (t');	963.11	char/ms
Record size;	470.00	bytes
File size;	47000000.00	bytes
T_F	24400.10	ms
T_N	24400.10	ms
T_I	41.24	ms
T_U	24457.94	ms
T_X (Serial read)	48800.20	ms
T_X (Sequential read)	626800.20	ms
T_Y	106384.00	ms

THE SEQUENTIAL FILE

The following data are required for sequential file model performance parameters measurements. parameter measurements. You have to enter the data in the file INPUT SEQ.

```
2               file type    (1 .. 5)   2 for sequential file
1024            block size (bytes) (1..99999)  (B)
100000          # of records in the main file (1..9999999) (n)
10              # of attributes per record (1..99)     (a)
47              attribute value length (bytes) (1..300) (V)
10000           # of inserts between reorganization (1..99999)
5000            # of deletes between reorganization (1..99999)
3000            # of updates between reorganization (1..99999)
1               type of storage device  (1..4)
                      1 for IBM 3380 Disk Pack
                      2 for IBM 2319 Disk Pack
                      3 for IBM 851 8" Floppy Disk
                      4 for IBM 9 Track Magnetic Tape
```

Example:

Consider the same example given for the pile file with the following changes.

a) Number of attributes per record = 10
b) Average attribute_value length = 47 bytes.

This makes record length = 470 bytes the same as that of pile file.

The program prints out the input data and then the seven measure of performance. The output should look like this

SEQUENTIAL FILE INPUT DATA

Block Size;	1,024	bytes
Total number of records in the main file;	100,000	records
# of attributes per record;	10	attr.
Average attribute_value Length;	47	bytes
Average # of inserts between reorganization	10,000	records
Average # of deletes between reorganization	5,000	records
Average # of updates between reorganization	4,000	records

SEQUENTIAL FILE PERFORMANCE PARAMETERS

Blocking factor;	2.00	records
Block transfer rate (btt);	0.34	ms
Bulk transfer rate (t');	963.11	char/ms
Record size;	470.00	bytes
File size;	47000000.00	bytes
Overflow file size;	9870000.00	bytes
T_F (Search argument is not the key);	29524.12	ms
T_F (Search argument is the key);	5508.64	ms
T_N	0.48	ms
T_I	51.58	ms
T_U	29575.70	ms
T_X	166100.00	ms
T_Y	217300.00	ms

THE INDEXED SEQUENTIAL FILE

The following data are required for Indexed Sequential file model performance parameters measurements. You have to enter the data in the file INPUT INDSEQ.

```
3               file type    (1 .. 5) 3 for index-sequential file
512             block size (bytes) (1..99999)    (B)
100000          # of records in the main file (1..9999999)  (n)
10              # of attributes per record (1..99)          (a)
15              attribute value length (bytes) (1..300)     (V)
4               pointer size(1..99)                          (P)
8000            # of inserts between reorganization (1..99999)
5000            # of deletes between reorganization (1..99999)
7000            # of updates between reorganization (1..99999)
2               TYPE OF STORAGE DEVICE  (1..4)
                      1 for IBM 3380 Disk Pack
                      2 for IBM 2319 Disk Pack
                      3 for IBM 851 8" Floppy Disk
                      4 for IBM 9 Track Magnetic Tape
```

Example:

Consider an indexed sequential file with a block size 512 bytes. The records are of fixed length with average value length of 15 bytes and 10 attributes per record. Allow a pointer size of 4 bytes. Pointer field is found in the record in the indexed level and in the records in the main file(They act as links to overflow records to maintain the sequence chain). Let the average number of inserts, deletes and updates be 8,000, 5,000 and 7,000 records.

Calculate the seven measure of performance.

The program prints out the input data and then the seven measure of performance. The output should look like this

INDEXED SEQUENTIAL FILE INPUT DATA

Block Size;	512	bytes
Total number of records in the main file;	100,000	records
# of attributes per record	10	
Average attribute_value length;	15	bytes
Pointer Size;	4	bytes
Average # of inserts between reorganization;	8,000	records
Average # of deletes between reorganization;	5,000	records
Average # of updates between reorganization;	7,000	records

INDEXED SEQUENTIAL FILE PERFORMANCE PARAMETERS

Blocking factor;	3.00	records
Block transfer rate (btt)	1.64	ms
Bulk transfer rate (t');	108.00	char/ms
Record size;	154.00	bytes
File size;	15400000.00	bytes
Overflow file size;	2310000.00	bytes
Index file size;	684032.00	bytes
T_F	134.33	ms
T_N	5.37	ms
T_I	198.47	ms
T_U	357.80	ms
T_X (Serial)	577403.96	ms
T_X (Sequential)	163981.48	ms
T_Y	740568.13	ms

THE INDEXED FILE

The following data are required for Indexed file model performance parameters measurements. Enter the data in the file INPUT IND

```
4               file type    (1 .. 5)
512             block size (bytes) (1..99999)   (B)
100000          # of records in the main file (1..9999999)   (n)
2               record format (1..2) 1 for fixed 2 for variable
10              avg # of attributes per record  (1 ..99)   (a')
14              total # of attributes in the file (1..99)  (a)
7               attribute name length  (bytes) (1..300)    (A)
6               attribute value length (bytes) (1..300)    (V)
6               index attribute length in index file (1..999) (V index)
4               pointer size (1..99)    (P)
69              density of index file(50..100)   (dens)
4               average # of attributes to be updated(1..99) (a update)
2               type of storage device (1..4)
                      1 for IBM 3380 Disk Pack
                      2 for IBM 2319 Disk Pack
                      3 for IBM 851 8" Floppy Disk
                      4 for IBM 9 Track Magnetic Tape
```

THE DIRECT FILE

The following data are required for Direct file model performance parameters measurements. Enter the data in file INPUT DIR

```
5               file type    (1 .. 5)
1024            block size (bytes) (1..99999)  (B)
100000          # of records in the main file (1..9999999)  (n)
150000          # of slots in the main file (1..9999999)    (m)
2               collision resolution (1..2) 1--Open addr. 2--chaining.
10              attributes per record  (1 ..999)  (a)
25              attribute value length (bytes) (1..999)  (V)
1               type of storage device  (1..4)
                      1 for IBM 3380 Disk Pack
                      2 for IBM 2319 Disk Pack
                      3 for IBM 851 8" Floppy Disk
                      4 for IBM 9 Track Magnetic Tape
6               pointer size (1..99)   (P) reqd. only on chained ovflw.
```

APPENDIX C

GENERIC SOFTWARE PACKAGE SOURCE CODE

```
*PROCESS OPTIONS MAR(2,72,1);
 (SIZE): CALC:
   PROCEDURE OPTIONS(MAIN);
 /*****************************************************/
 /*              MAIN PROCEDURE                       */
 /*****************************************************/
  DECLARE
    (A_PRIME,
    A,
    A_UPD,
    ATTR_LEN,
    VALUE_SZ,
    BLOCK_SZ,
    VALUE_INDX_SZ)      FIXED DECIMAL(11,2),

    (REC_SZ,
    BFR)                FIXED DECIMAL(9,2),
   (PTR_SZ,
    N,
    M)                  FIXED DECIMAL(11,2),
    (OV_FLW_SZ,
    NO_INSRT,
    NO_DLTE,
    NO_UPDTE,COLL_TYPE)  FIXED DECIMAL(11,2),
    FILE_TYPE           FIXED DECIMAL(2),
    DISP_TYPE           FIXED DEC(1);
  DECLARE (REC_FRMT,
           DISK_TYPE)    FIXED DEC(11,2);
 DECLARE
   WASTE               FIXED DECIMAL(9,2) ,
    TRW                FIXED DECIMAL(8,2),
    BTT                FIXED DECIMAL(9,2),
    T_PRIME            FIXED DECIMAL(7,2);

  DECLARE
       S                FIXED DECIMAL(10,2),
       R                FIXED DECIMAL(8,2),
       GAP              FIXED DECIMAL(8,2),
       TRNSFR_RTE       FIXED DECIMAL(8,2),
       DENSITY          FIXED DECIMAL(11,2),
       SIZEF            BIT(1);
 DECLARE INFILE STREAM FILE INPUT,
         SYSIN  STREAM FILE INPUT,
         SYSPRINT PRINT FILE OUTPUT,
         (LOG,LOG2,DEC,ABS,EXP,FLOOR,CEIL,ONSOURCE) BUILTIN;
 DCL   FLAG           CHAR(3);

 ON ERROR
   BEGIN;
```

```
    ON ERROR SYSTEM;
    IF FLAG = '1' THEN
      PUT FILE(SYSPRINT) LIST ('RECORD SIZE OF A SINGLE RECORD '||
         'EXCEEDS BLOCK SIZE. THIS IS NOT POSSIBLE WITH UNSPANNED'||
         ' BLOCKING');
    IF FLAG = '2' THEN
     PUT FILE(SYSPRINT) LIST (' INPUT VALUE OUT OF RANGE');
    IF FLAG = '3' THEN
     PUT FILE(SYSPRINT) LIST (' # OF RECORDS + # NO OF INSERTS '||
       '+ # OF DELETES IS NEGATIVE. ERROR');
    IF FLAG = '4' THEN
     PUT FILE(SYSPRINT) LIST ('# OF UPDATE ATTRIBUTES EXCEEDS '||
       'TOTAL # OF ATTRIBUTES .');
    IF FLAG = '5' THEN
     PUT FILE(SYSPRINT) LIST ('VALUE SIZE AND POINTER SIZE EXCEEDS '||
       'BLOCK SIZE. INDEX FOR A RECORD CANNOT FIT IN A BLOCK.');
    IF FLAG = '6' THEN
     PUT FILE(SYSPRINT) LIST ('VALUE SIZE OF INDEX AND POINTER SIZE'||
' EXCEEDS BLOCK SIZE. INDEX FOR A RECORD CANNOT FIT IN A BLOCK.');
  END;
ON CONVERSION BEGIN;
  PUT FILE(SYSPRINT) LIST ('INVALID NUMERIC FIELD =',ONSOURCE);
  EXIT;
END;
GET LIST (FILE_TYPE);
SELECT (FILE_TYPE);
  WHEN (1) CALL  ENTER_PILE_DATA;
  WHEN (2) CALL  ENTER_SEQUEN_DATA;
  WHEN (3) CALL  ENTER_ISEQ_DATA;
  WHEN (4) CALL  ENTER_INDXD_DATA;
  WHEN (5) CALL  ENTER_DIRECT_DATA;
  OTHERWISE DO;
              PUT SKIP LIST ('WRONG FILE_TYPE ENTERED');  EXIT;
            END;
END;
CALL DISK_PARM;
CALL RECORD_SZ;
IF (FILE_TYPE = 2) | (FILE_TYPE = 3) | (FILE_TYPE = 5) THEN
  REC_FRMT = 1;
IF (FILE_TYPE = 1) THEN REC_FRMT = 2;
CALL BLKING_FACT_CALC((REC_FRMT),(BLOCK_SZ),BFR,(GAP),WASTE);
CALL BLOCK_TRNSFR_RTE_CALC(BTT,TRW,(R),(BLOCK_SZ),(TRNSFR_RTE));
CALL T_PRIME_CALC((REC_SZ),(WASTE),T_PRIME);
SELECT (FILE_TYPE);
    WHEN (1) DO;
      CALL DISPLAY_PILE_DATA;
      CALL PILE;
    END;
    WHEN (2) DO;
      CALL DISPLAY_SEQ_DATA;
      CALL SEQUENTIAL;
    END;
    WHEN (3) DO;
      CALL DISPLAY_INDSEQ_DATA;
      CALL INDEXED_SEQUENTIAL;
```

```
    END;
    WHEN (4) DO;
      CALL DISPLAY_INDXD_DATA;
      CALL INDEXED;
    END;
    WHEN (5) DO;
      CALL DISPLAY_DIRECT_DATA;
      CALL DIRECT;
    END;
    OTHERWISE
      PUT LIST ('ERROR IN FILE TYPE');
  END;
/*************************************************
**     READ INPUT DATA FOR PILE FILE          **
*************************************************/
ENTER_PILE_DATA : PROCEDURE;

 GET FILE(SYSIN) EDIT(BLOCK_SZ) (COL(1), F(6));
 CALL VALID(1,99999,BLOCK_SZ,'BLOCK SIZE ' );
 GET FILE(SYSIN) EDIT(N) (COL(1),F(8));
 CALL VALID(1,9999999,N,' NO OF RECORDS');
 GET FILE(SYSIN) EDIT (PTR_SZ)(COL(1),F(3));
 CALL VALID (1,99,PTR_SZ,' POINTER SIZE');
 GET FILE(SYSIN) EDIT(A_PRIME) (COL(1),F(4));
 CALL VALID (1,99,A_PRIME ,' AVERAGE # OF ATTRIBUTES');
 GET FILE(SYSIN) EDIT(ATTR_LEN) (COL(1),F(4));
 CALL VALID (1,300,ATTR_LEN,' ATTRIBUTE_NAME LENGTH');
 GET FILE(SYSIN) EDIT (VALUE_SZ) (COL(1),F(4));
 CALL VALID (1,300,VALUE_SZ,' ATTRIBUTE_VALUE LENGTH');
 GET FILE(SYSIN) EDIT(NO_INSRT) (COL(1),F(6));
 CALL VALID (1,99999,NO_INSRT,' # OF INSERTS');
 GET FILE(SYSIN) EDIT(NO_DLTE) (COL(1),F(6));
 CALL VALID (1,99999,NO_DLTE,' # OF DELETES');
 GET FILE(SYSIN) EDIT(NO_UPDTE) (COL(1),F(6));
 CALL VALID (1,99999,NO_UPDTE,' # OF UPDATES');
 GET FILE (SYSIN) EDIT(DISK_TYPE) (COL(1),F(2));
 CALL VALID(1,4,DISK_TYPE,'DISK TYPE');
END ENTER_PILE_DATA;
/*************************************************
**     READ INPUT DATA FOR SEQUENTIAL FILE    **
*************************************************/
ENTER_SEQUEN_DATA : PROCEDURE;
 GET FILE(SYSIN) EDIT(BLOCK_SZ) (COL(1), F(6));
 CALL VALID(1,99999,BLOCK_SZ,'BLOCK SIZE ' );
 GET FILE(SYSIN) EDIT(N) (COL(1),F(8));
 CALL VALID(1,9999999,N,' NO OF RECORDS');
 GET FILE(SYSIN) EDIT(A) (COL(1),F(4));
 CALL VALID (1,99,A,' TOTAL # OF ATTRIBUTES');
 GET FILE(SYSIN) EDIT (VALUE_SZ) (COL(1),F(4));
 CALL VALID (1,300,VALUE_SZ,' ATTRIBUTE_VALUE LENGTH');
 GET FILE(SYSIN) EDIT(NO_INSRT) (COL(1),F(6));
 CALL VALID (1,99999,NO_INSRT,' # OF INSERTS');
 GET FILE(SYSIN) EDIT(NO_DLTE) (COL(1),F(6));
 CALL VALID (1,99999,NO_DLTE,' # OF DELETES');
 GET FILE(SYSIN) EDIT(NO_UPDTE) (COL(1),F(6));
```

```
 CALL VALID (1,99999,NO_UPDTE,' # OF UPDATES');
GET FILE (SYSIN) EDIT(DISK_TYPE) (COL(1),F(2));
 CALL VALID(1,4,DISK_TYPE,'DISK TYPE');
END ENTER_SEQUEN_DATA;
/*********************************************
**    READ INPUT DATA FOR INDEXED SEQ FILE  **
*********************************************/
ENTER_ISEQ_DATA : PROCEDURE;
 GET FILE(SYSIN) EDIT(BLOCK_SZ) (COL(1), F(6));
 CALL VALID(1,99999,BLOCK_SZ,'BLOCK SIZE ' );
 GET FILE(SYSIN) EDIT(N) (COL(1),F(8));
 CALL VALID(1,9999999,N,' NO OF RECORDS');
 GET FILE(SYSIN) EDIT(A) (COL(1),F(4));
 CALL VALID (1,99,A,' TOTAL # OF ATTRIBUTES');
 GET FILE(SYSIN) EDIT (VALUE_SZ) (COL(1),F(4));
 CALL VALID (1,300,VALUE_SZ,' ATTRIBUTE_VALUE LENGTH');
 GET FILE(SYSIN) EDIT (PTR_SZ)(COL(1),F(3));
 CALL VALID (1,99,PTR_SZ,' POINTER SIZE');
 GET FILE(SYSIN) EDIT(NO_INSRT) (COL(1),F(6));
 CALL VALID (1,99999,NO_INSRT,' # OF INSERTS');
 GET FILE(SYSIN) EDIT(NO_DLTE) (COL(1),F(6));
 CALL VALID (1,99999,NO_DLTE,' # OF DELETES');
 GET FILE(SYSIN) EDIT(NO_UPDTE) (COL(1),F(6));
 CALL VALID (1,99999,NO_UPDTE,' # OF UPDATES');
 GET FILE (SYSIN) EDIT(DISK_TYPE) (COL(1),F(2));
 CALL VALID(1,4,DISK_TYPE,'DISK TYPE');
END ENTER_ISEQ_DATA;
/*********************************************
**    READ INPUT DATA FOR INDEXED FILE     **
*********************************************/
ENTER_INDXD_DATA : PROCEDURE;
 GET FILE(SYSIN) EDIT(BLOCK_SZ) (COL(1), F(6));
 CALL VALID(1,99999,BLOCK_SZ,'BLOCK SIZE ' );
 GET FILE(SYSIN) EDIT(N) (COL(1),F(8));
 CALL VALID(1,9999999,N,' NO OF RECORDS');
 GET FILE (SYSIN) EDIT(REC_FRMT) (COL(1),F(2));
 CALL VALID (1,2,REC_FRMT,' RECORD FORMAT');
 GET FILE(SYSIN) EDIT(A_PRIME) (COL(1),F(4));
 CALL VALID (1,99,A_PRIME,' AVERAGE # OF ATTRIBUTES');
 GET FILE(SYSIN) EDIT(A) (COL(1),F(4));
 CALL VALID (1,99,A,' TOTAL # OF ATTRIBUTES');
 GET FILE(SYSIN) EDIT(ATTR_LEN) (COL(1),F(4));
 CALL VALID (1,300,ATTR_LEN,' ATTRIBUTE_NAME LENGTH');
 GET FILE(SYSIN) EDIT (VALUE_SZ) (COL(1),F(4));
 CALL VALID (1,300,VALUE_SZ,' ATTRIBUTE_VALUE LENGTH');
 GET FILE(SYSIN) EDIT (VALUE_INDX_SZ) (COL(1),F(4));
 CALL VALID (1,999,VALUE_INDX_SZ,' ATTRIBUTE_VALUE LENGTH');
 GET FILE(SYSIN) EDIT (PTR_SZ) (COL(1),F(3));
 CALL VALID (1,99,PTR_SZ,' POINTER SIZE');
 GET FILE(SYSIN) EDIT (DENSITY) (COL(1),F(4));
 CALL VALID (50,100,DENSITY,' DENSITY ');
 GET FILE(SYSIN) EDIT(A_UPD) (COL(1),F(4));
 CALL VALID (1,999,A_UPD,' ATTRIBUTE UPDATE');
GET FILE (SYSIN) EDIT(DISK_TYPE) (COL(1),F(2));
 CALL VALID(1,4,DISK_TYPE,'DISK TYPE');
```

```
END ENTER_INDXD_DATA;
/*********************************************
**     READ INPUT DATA FOR DIRECT FILE       **
*********************************************/
ENTER_DIRECT_DATA : PROCEDURE;
 GET FILE(SYSIN) EDIT(BLOCK_SZ) (COL(1), F(6));
 CALL VALID(1,99999,BLOCK_SZ,'BLOCK SIZE ' );
 GET FILE(SYSIN) EDIT(N) (COL(1),F(8));
 CALL VALID(1,9999999,N,' NO OF RECORDS');
 GET FILE(SYSIN) EDIT(M) (COL(1),F(8));
 CALL VALID(1,9999999,M,' NO OF RECORDS');
 GET FILE (SYSIN) EDIT(COLL_TYPE) (COL(1),F(2));
 CALL VALID (1,2,COLL_TYPE,' COLLISION TYPE');
 GET FILE(SYSIN) EDIT(A) (COL(1),F(4));
 CALL VALID (1,99,A,' TOTAL # OF ATTRIBUTES');
 GET FILE(SYSIN) EDIT (VALUE_SZ) (COL(1),F(4));
 CALL VALID (1,300,VALUE_SZ,' ATTRIBUTE_VALUE LENGTH');
 GET FILE (SYSIN) EDIT(DISK_TYPE) (COL(1),F(2));
 CALL VALID(1,4,DISK_TYPE,'DISK TYPE');
 IF COLL_TYPE = 2 THEN
   DO;
     GET FILE(SYSIN) EDIT (PTR_SZ) (SKIP(5),COL(1),F(3));
     CALL VALID (1,99,PTR_SZ,' POINTER SIZE');
   END;
END ENTER_DIRECT_DATA;
/*****************************************
**      CHECK RANGE OF INPUT DATA      **
*****************************************/
VALID: PROCEDURE (LBOUND,HBOUND,VALUE,VAR);
DCL   (HBOUND,
       VALUE,
       LBOUND)    FIXED DECIMAL(11,2),
       VAR        CHAR(*) ;
IF (VALUE < LBOUND) | ( VALUE > HBOUND) THEN
  DO;
    FLAG = '2';
    PUT LIST (VAR);
    SIGNAL ERROR;
  END;
END;
/*************************************************
**     GET DISK PARAMETERS FROM FILE             **
*************************************************/
DISK_PARM : PROCEDURE;
 DECLARE  TEMP          FIXED DEC(2,0);
GET FILE (INFILE) EDIT (TEMP) (COL(1),F(1,0));
DO WHILE(TEMP^=DISK_TYPE
  GET FILE (INFILE) EDIT (TEMP) (COL(1),F(1,0));
END;
GET FILE(INFILE) EDIT (S,R,GAP,TRNSFR_RTE) (COL(3),F(10,2),
          COL(15),F(8,2),COL(25),F(8,2),COL(35),F(8,2));
END DISK_PARM;
/*****************************************
**     DISPLAY PILE INPUT DATA           **
*****************************************/
```

```
DISPLAY_PILE_DATA : PROCEDURE;
 PUT SKIP(2) FILE(SYSPRINT) EDIT('PILE FILE INPUT DATA') (COL(20),A);
 PUT SKIP(0) FILE(SYSPRINT) EDIT('____________________') (COL(20),A);
PUT FILE(SYSPRINT) SKIP(3) EDIT('BLOCK SIZE: ',BLOCK_SZ,'BYTES')
   (COL(10),A,COL(60),P'ZZZ,ZZ9',X(1),A);
 PUT SKIP(2) FILE(SYSPRINT) EDIT('TOTAL NUMBER OF RECORDS ',
  'IN THE MAIN FILE:',N,'RECORDS') (COL(10),A,A,COL(60),P'Z,ZZZ,ZZ9',
     X(1),A);
IF (REC_FRMT = 1) THEN
PUT SKIP(2)FILE(SYSPRINT) EDIT('RECORD TYPE:','FIXED LENGTH RECORDS')
  (COL(10),A,COL(65),A);
ELSE
 PUT SKIP(2)FILE(SYSPRINT) EDIT('RECORD TYPE:',
  'VARYING LENGTH RECORDS') (COL(10),A,COL(65),A);
 IF REC_FRMT = 2 THEN
   PUT SKIP(2) FILE(SYSPRINT) EDIT(' POINTER SIZE: ',PTR_SZ,'BYTES')
     (COL(10),A,COL(60),F(3),X(1),A);
PUT SKIP(2) FILE(SYSPRINT) EDIT('AVERAGE # OF ATTRIBUTES PER RECORD:',
   A_PRIME) (COL(10),A,COL(60),P'ZZZ,ZZ9');
 PUT FILE(SYSPRINT) EDIT('AVERAGE ATTRIBUTE_NAME LENGTH:',ATTR_LEN,
'BYTES') (SKIP(2),COL(10),A,COL(60),P'ZZZ,ZZ9',X(1),A);
 PUT FILE(SYSPRINT) EDIT('AVERAGE ATTRIBUTE_VALUE LENGTH:',VALUE_SZ,
'BYTES') (SKIP(2),COL(10),A,COL(60),P'ZZZ,ZZ9',X(1),A);
 PUT FILE(SYSPRINT) EDIT ('AVERAGE NUMBER OF INSERTS BETWEEN '||
 'REORGANIZATION',NO_INSRT) (SKIP(2),COL(10),A,COL(60),P'ZZZ,ZZ9');
 PUT FILE(SYSPRINT) EDIT ('AVERAGE NUMBER OF DELETES BETWEEN '||
 'REORGANIZATION',NO_DLTE) (SKIP(2),COL(10),A,COL(60),P'ZZZ,ZZ9');
 PUT FILE(SYSPRINT) EDIT ('AVERAGE NUMBER OF UPDATES BETWEEN '||
 'REORGANIZATION',NO_UPDTE) (SKIP(2),COL(10),A,COL(60),P'ZZZ,ZZ9');
PUT PAGE FILE(SYSPRINT);
END DISPLAY_PILE_DATA;
/*****************************************************
**   DISPLAY SEQUENTIAL FILE INPUT DATA          **
*****************************************************/
DISPLAY_SEQ_DATA : PROCEDURE;
 PUT SKIP(2) FILE(SYSPRINT) EDIT('SEQUENTIAL FILE INPUT DATA')
(COL(20),A);
 PUT SKIP(0) FILE(SYSPRINT) EDIT('____________________________')
(COL(20),A);
 PUT FILE(SYSPRINT) SKIP(3) EDIT('BLOCK SIZE: ',BLOCK_SZ,'BYTES')
   (COL(10),A,COL(60),P'ZZZ,ZZ9',X(1),A);
 PUT FILE(SYSPRINT) SKIP(2) EDIT('TOTAL NUMBER OF RECORDS ',
  'IN THE MAIN FILE:',N,'RECORDS') (COL(10),A,A,COL(60),P'Z,ZZZ,ZZ9',
     X(3),A);
 PUT FILE(SYSPRINT) SKIP(2) EDIT(' # OF ATTRIBUTES PER RECORD',
   A) (COL(10),A,COL(60),P'ZZZ,ZZ9');
 PUT FILE(SYSPRINT) EDIT('AVERAGE ATTRIBUTE_VALUE LENGTH:',VALUE_SZ,
'BYTES') (SKIP(2),COL(10),A,COL(60),P'ZZZ,ZZ9',X(3),A);
 PUT FILE(SYSPRINT) EDIT ('AVERAGE NUMBER OF INSERTS BETWEEN '||
 'REORGANIZATION',NO_INSRT) (SKIP(2),COL(10),A,COL(60),P'ZZZ,ZZ9');
 PUT FILE(SYSPRINT) EDIT ('AVERAGE NUMBER OF DELETES BETWEEN '||
 'REORGANIZATION',NO_DLTE) (SKIP(2),COL(10),A,COL(60),P'ZZZ,ZZ9');
 PUT FILE(SYSPRINT) EDIT ('AVERAGE NUMBER OF UPDATES BETWEEN '||
 'REORGANIZATION',NO_UPDTE) (SKIP(2),COL(10),A,COL(60),P'ZZZ,ZZ9');
PUT PAGE FILE(SYSPRINT);
```

```
END DISPLAY_SEQ_DATA;
/*************************************************
**   DISPLAY IND  SEQ FILE INPUT DATA            **
*************************************************/
DISPLAY_INDSEQ_DATA : PROCEDURE;
 PUT SKIP(2) FILE(SYSPRINT) EDIT('INDEXED SEQUENTIAL FILE INPUT DATA')
(COL(20),A);
 PUT SKIP(0) FILE(SYSPRINT) EDIT('_____________________________________')
(COL(20),A);
 PUT FILE(SYSPRINT) SKIP(3) EDIT('BLOCK SIZE: ',BLOCK_SZ,'BYTES')
   (COL(10),A,COL(60),P'ZZZ,ZZ9',X(1),A);
 PUT FILE(SYSPRINT) SKIP(2)  EDIT('TOTAL NUMBER OF RECORDS ',
  'IN THE MAIN FILE:',N,'RECORDS') (COL(10),A,A,COL(60),P'Z,ZZZ,ZZ9',
     X(1),A);
 PUT FILE(SYSPRINT) SKIP(2) EDIT(' # OF ATTRIBUTES PER RECORD',
   A) (COL(10),A,COL(60),P'ZZZ,ZZ9');
 PUT FILE(SYSPRINT) EDIT('AVERAGE ATTRIBUTE_VALUE LENGTH:',VALUE_SZ,
'BYTES') (SKIP(2),COL(10),A,COL(60),P'ZZZ,ZZ9',X(3),A);
 PUT FILE(SYSPRINT) SKIP(2) EDIT(' POINTER SIZE: ',PTR_SZ,'BYTES')
   (COL(10),A,COL(60),F(3),X(3),A);
 PUT FILE(SYSPRINT) SKIP(2)EDIT ('AVERAGE NUMBER OF INSERTS '||
 'BETWEEN REORGANIZATION',NO_INSRT) (COL(10),A,COL(60),P'ZZZ,ZZ9');
 PUT FILE(SYSPRINT) SKIP(2) EDIT ('AVERAGE NUMBER OF DELETES '||
 'BETWEEN REORGANIZATION',NO_DLTE) (COL(10),A,COL(60),P'ZZZ,ZZ9');
 PUT FILE(SYSPRINT) SKIP(2) EDIT ('AVERAGE NUMBER OF UPDATES '||
 'BETWEEN REORGANIZATION',NO_UPDTE) (COL(10),A,COL(60),P'ZZZ,ZZ9');
PUT PAGE FILE(SYSPRINT);
END DISPLAY_INDSEQ_DATA;
/*****************************************
**    DISPLAY INDEXED INPUT DATA         **
*****************************************/
DISPLAY_INDXD_DATA : PROCEDURE;
 PUT SKIP(2) FILE(SYSPRINT) EDIT('INDEXED FILE INPUT DATA')
(COL(20),A);
 PUT SKIP(0) FILE(SYSPRINT) EDIT('_____________________')
(COL(20),A);
 PUT FILE(SYSPRINT) SKIP(3) EDIT('BLOCK SIZE: ',BLOCK_SZ,'BYTES')
   (COL(10),A,COL(55),P'ZZZ,ZZ9',X(1),A);
 PUT FILE(SYSPRINT) SKIP(2) EDIT('TOTAL NUMBER OF RECORDS ',
  'IN THE MAIN FILE:',N,'RECORDS') (COL(10),A,A,COL(55),P'Z,ZZZ,ZZ9',
     X(1),A);
IF REC_FRMT = 1 THEN
  PUT FILE(SYSPRINT) EDIT('RECORD TYPE:','FIXED LENGTH RECORDS ')
  (SKIP(2),COL(10),A,COL(55),A);
ELSE
 PUT FILE(SYSPRINT) EDIT('RECORD TYPE:','VARYING LENGTH RECORDS ')
  (SKIP(2),COL(10),A,COL(55),A);
   PUT FILE(SYSPRINT) SKIP(2) EDIT(' POINTER SIZE: ',PTR_SZ,'BYTES')
     (COL(10),A,COL(55),F(3),X(1),A);
 PUT FILE(SYSPRINT) SKIP(2) EDIT('AVERAGE # OF ATTRIBUTES PER RECORD:',
   A_PRIME) (COL(10),A,COL(55),P'ZZZ,ZZ9');
 PUT FILE(SYSPRINT) SKIP(2) EDIT('TOTAL # OF ATTRIBUTES IN THE FILE:',
   A) (COL(10),A,COL(55),P'ZZZ,ZZ9');
 PUT FILE(SYSPRINT) EDIT('AVERAGE ATTRIBUTE_NAME LENGTH:',ATTR_LEN,
'BYTES') (SKIP(2),COL(10),A,COL(55),P'ZZZ,ZZ9',X(3),A);
```

```
 PUT FILE(SYSPRINT) EDIT('AVERAGE ATTRIBUTE_VALUE LENGTH:',VALUE_SZ,
'BYTES') (SKIP(2),COL(10),A,COL(55),P'ZZZ,ZZ9',X(3),A);
 PUT FILE(SYSPRINT) SKIP(2) EDIT('AVERAGE LENGTH OF INDEX ATTRIBUTE:',
  VALUE_INDX_SZ,'BYTES') (COL(10),A,COL(55),P'ZZZ,ZZ9',X(3),A);
 PUT FILE(SYSPRINT) SKIP(2) EDIT('AVERAGE # OF UPDATE ATTRIBUTES: ',
  A_UPD) (COL(10),A,COL(55),P'ZZZ,ZZ9');
 PUT FILE(SYSPRINT) SKIP(2) EDIT('DENSITY OF INDEX FILE: ',
  DENSITY,'%') (COL(10),A,COL(55),P'ZZZ,ZZ9',X(1),A);
PUT PAGE FILE(SYSPRINT);
END DISPLAY_INDXD_DATA;
/*****************************************
**    DISPLAY DIRECT  INPUT DATA         **
*****************************************/
DISPLAY_DIRECT_DATA : PROCEDURE;
 PUT SKIP(2) FILE(SYSPRINT) EDIT('DIRECT FILE INPUT DATA')
(COL(20),A);
 PUT SKIP(0) FILE(SYSPRINT) EDIT('_____________________')
(COL(20),A);
 PUT FILE(SYSPRINT) SKIP(3) EDIT('BUCKET SIZE: ',BLOCK_SZ,'BYTES')
   (COL(10),A,COL(55),P'ZZZ,ZZ9',X(1),A);
 PUT FILE(SYSPRINT) SKIP(2) EDIT('TOTAL NUMBER OF RECORDS ',
  'IN THE MAIN FILE:',N,'RECORDS') (COL(10),A,A,COL(55),P'Z,ZZZ,ZZ9',
     X(1),A);
 PUT FILE(SYSPRINT) EDIT('TOTAL NUMBER OF SLOTS FOR THE RECORDS '||
  'IN THE MAIN FILE:',M,'RECORDS') (SKIP(2),COL(10),A, COL(55),
     P'Z,ZZZ,ZZ9',X(1),A);
IF COLL_TYPE= 1 THEN
PUT FILE(SYSPRINT) EDIT('COLLISION RESOLUTION TYPE','OPEN ADDRESSING')
  (SKIP(2),COL(10),A,COL(55),A);
ELSE
PUT FILE(SYSPRINT) EDIT('COLLISION RESOLUTION TYPE:','CHAINED ACCESS')
  (SKIP(2),COL(10),A,COL(55),A);
PUT FILE(SYSPRINT) EDIT('# OF ATTRIBUTES PER RECORD:',
   A) (SKIP(2),COL(10),A,COL(55),P'ZZZ,ZZ9');
 PUT FILE(SYSPRINT) EDIT('AVERAGE ATTRIBUTE_VALUE LENGTH:',VALUE_SZ,
'BYTES') (SKIP(2),COL(10),A,COL(55),P'ZZZ,ZZ9',X(1),A);
IF COLL_TYPE = 2 THEN
   PUT FILE(SYSPRINT) EDIT(' POINTER SIZE: ',PTR_SZ,'BYTES')
     (SKIP(2),COL(10),A,COL(55),F(3),X(1),A);
END DISPLAY_DIRECT_DATA;
/*****************************************/
/*   DISPLAY BTT,BFR,T_PRIME VALUES      */
/*****************************************/
DISP_COMMON_DATA: PROCEDURE;
FMT: FORMAT(SKIP(2),COL(10),A,COL(55),F(15,2),X(1),A);
PUT SKIP(1) EDIT ('BLOCKING FACTOR: ',BFR,'RECORDS') (R(FMT));
PUT EDIT ('BLOCK TRANSFER RATE (BTT): ',BTT,'MS') (R(FMT));
PUT EDIT('BULK TRANSFER RATE (T PRIME): ',T_PRIME,'CHAR/MS') (R(FMT));
END DISP_COMMON_DATA;
/*****************************************/
/*         RECORD SIZE MODULE            */
/*****************************************/
RECORD_SZ: PROCEDURE;
IF (FILE_TYPE = 1) | (FILE_TYPE = 4)
THEN
```

```
  REC_SZ = A_PRIME * (ATTR_LEN + VALUE_SZ + 2);
IF (FILE_TYPE = 2) | (FILE_TYPE = 3)
THEN
  REC_SZ =  A * VALUE_SZ;
IF (FILE_TYPE = 5) THEN
  IF COLL_TYPE = 1 THEN
  REC_SZ =  A *  VALUE_SZ ;
  ELSE
  REC_SZ = A * VALUE_SZ + PTR_SZ;
END RECORD_SZ;
/*********************************************/
/*     BLOCK TRANSFER RATE MODULE            */
/*********************************************/
BLOCK_TRNSFR_RTE_CALC:
    PROCEDURE(BTT,TRW,R,BLOCK_SZ,TRNSFR_RTE);
  DECLARE
    BTT                FIXED DECIMAL(9,2),
    BLOCK_SZ           FIXED DECIMAL(7),
    TRW                FIXED DECIMAL(8,2),
    TRNSFR_RTE         FIXED DECIMAL(8,2),
    R                  FIXED DECIMAL(8,2);
  BTT = BLOCK_SZ / TRNSFR_RTE;
  TRW = 2 * R ; /* FOR NOW*/
  END BLOCK_TRNSFR_RTE_CALC;

  /*************************************************/
  /*       BLOCKING FACTOR MODULE                  */
  /*************************************************/
  BLKING_FACT_CALC:
     PROCEDURE(REC_FRMT,BLOCK_SZ,BFR,GAP,WASTE);

  DECLARE
    REC_FRMT     FIXED DECIMAL(1),
    BFR          FIXED DECIMAL(9,2),
    BLOCK_SZ     FIXED DECIMAL(7),
    WASTE        FIXED DECIMAL(9,2),
    GAP          FIXED DECIMAL(4);

  IF REC_FRMT =  1  THEN DO;
   BFR = FLOOR(BLOCK_SZ / REC_SZ);
   IF BFR <= 0 THEN
    DO;
     FLAG = '1';
     SIGNAL ERROR;
    END;
   WASTE = GAP / BFR;
  END;
  ELSE DO;
   BFR = FLOOR((BLOCK_SZ - (0.5 * REC_SZ)) / (REC_SZ + PTR_SZ));
   IF BFR <=  0 THEN
     DO;
       FLAG = '1';
       SIGNAL ERROR;
     END;
   WASTE  =  PTR_SZ + (0.5 * REC_SZ + GAP) /BFR;
```

```
   END;
END BLKING_FACT_CALC;

/*********************************************************/
/*        BULK TRANSFER TIME MODULE                      */
/*********************************************************/
T_PRIME_CALC:
 PROCEDURE(REC_SZ,WASTE,T_PRIME);
 DECLARE
   REC_SZ       FIXED DECIMAL(6),
   WASTE        FIXED DECIMAL(5,2),
   T_PRIME  FIXED DECIMAL(7,2);

   T_PRIME = 0.5 * TRNSFR_RTE * (REC_SZ / (REC_SZ + WASTE)) ;
END T_PRIME_CALC;

/*********************************************************/
/*                SORT MODULE                            */
/*********************************************************/
 SORT: PROCEDURE(NN,BFR,BTT)
       RETURNS (FIXED DECIMAL (15,2));
DECLARE NN          FIXED DECIMAL(11,2),
       BFR          FIXED DECIMAL(9,2),
       B            FIXED DECIMAL(8),
       BTT          FIXED DECIMAL(9,2);
DECLARE N           FIXED DECIMAL(8);
N = NN;
B = CEIL(N / BFR);
RETURN ( 2 * B * BTT * (1 + CEIL(LOG2 (B))));
END SORT;
/*********************************************************
**          PILE FILE CALCULATIONS                      **
*********************************************************/
PILE: PROCEDURE;
DECLARE
  (TF,
   TN,
   TI,
   TU,
   TXSE,
   TXSQ,
   TY,
   REC_SZ,
   FILE_SZ)              FIXED DECIMAL (15,2);
DECLARE  O               FIXED DEC(7),
         D               FIXED DEC(7),
         NNEW            FIXED DEC(7);
REC_SZ = A_PRIME * ( ATTR_LEN + VALUE_SZ + 2);
FILE_SZ = N * REC_SZ;
TF = 0.5 * N * REC_SZ / T_PRIME;
TN = TF;
TI = S + R + BTT + TRW;
TU = TF + TRW + TI;
TXSE = 2 * TF;
TXSQ = SORT(N,BFR,BTT) + TXSE;
```

```
IF ((N + NO_INSRT- NO_DLTE) < 1) THEN
  DO;
   FLAG = '4';
   SIGNAL ERROR;
  END;
O = NO_INSRT + NO_UPDTE;
D = NO_UPDTE + NO_DLTE;
TY =  (N + O + N + O - D) * REC_SZ / T_PRIME;
   CALL PILE_DISP;
  RETURN;
/*********************************************
**      PILE RESULTS DISPLAY                 **
*********************************************/
PILE_DISP: PROCEDURE;
PUT SKIP(2) EDIT ('PILE FILE PERFORMANCE PARAMETERS') (COL(20),A);
PUT SKIP(0) EDIT ('_____________________________________') (COL(20),A);
CALL DISP_COMMON_DATA;
RFMT: FORMAT(COL(10),A,COL(55),F(15,2),X(1),A);
PUT SKIP(2) EDIT ('RECORD SIZE: ',REC_SZ,'BYTES') (R(RFMT));
PUT SKIP(2) EDIT ('FILE SIZE: ',FILE_SZ,'BYTES') (R(RFMT));
PUT SKIP(2) EDIT ('TF ',TF,'MS') (R(RFMT));
PUT SKIP(2) EDIT ('TN',TN,'MS') (R(RFMT));
PUT SKIP(2) EDIT ('TI',TI,'MS') (R(RFMT));
PUT SKIP(2) EDIT ('TU',TU,'MS') (R(RFMT));
PUT SKIP(2) EDIT ('TX (SEQUENTIAL READ)',TXSE,'MS') (R(RFMT));
PUT SKIP(2) EDIT ('TX (SERIAL READ)',TXSQ,'MS')
  (R(RFMT));
PUT SKIP(2) EDIT ('TY',TY,'MS') (R(RFMT));
END PILE_DISP;
END PILE;
/*******************************************************
*       SEQUENTIAL FILE CALCULATIONS                  **
*******************************************************/
SEQUENTIAL: PROCEDURE;
  DECLARE
   (TF1,
    TF2,
    TN ,
    TU,
    TI,
    TX,
    TY,
    FILE_SZ,
    O_FILE_SZ)                FIXED DEC(15,2);
 declare
   O_N                        FIXED DECIMAL (11,2),
   N_NEW                      FIXED DECIMAL  (9),
   REC_SZ                     FIXED DECIMAL (7);
REC_SZ = A * VALUE_SZ;
FILE_SZ =  N * REC_SZ;
IF ((N + NO_INSRT - NO_DLTE) < 1) THEN
  DO;
   SIZEF = '1'B;
   SIGNAL ERROR;
  END;
```

```
O_N = NO_INSRT + NO_DLTE + 2 * NO_UPDTE;
O_FILE_SZ = O_N * REC_SZ;
/* WHEN SEARCH ARGUMENT IS NOT THE KEY  */
TF1 = 0.5 * (N + O_N) * (REC_SZ / T_PRIME);
 /* WHEN SEARCH ARGUMENT IS THE KEY */
TF2 = (LOG2(N / BFR) * (S + R + BTT)) + 0.5 * (O_N) *
      (REC_SZ / T_PRIME);
TN = REC_SZ / T_PRIME;
TX = SORT(O_N,BFR,BTT) + ((N + O_N) * REC_SZ / T_PRIME);
N_NEW = N + NO_INSRT - NO_DLTE;
TY = SORT(O_N,BFR,BTT) + (N + O_N + N_NEW ) * REC_SZ / T_PRIME;
/* INSERTION TAKES PLACE IN THE LOG FILE ONLY. */
TI = S + R + BTT + TRW + TY / O_N ;
TU = TF1 + TI;
 CALL SEQ_RES_DISP;
 RETURN;
/*********************************************************
**       SEQUENTIAL FILE RESULTS DISPLAY               **
*********************************************************/
SEQ_RES_DISP: PROCEDURE;
RFMT: FORMAT(COL(10),A,COL(55),F(15,2),X(1),A);
PUT SKIP(2) EDIT ('SEQUENTIAL FILE PERFORMANCE PARAMETERS')
   (COL(20),A);
PUT SKIP(0) EDIT ('_____________________________________')
   (COL(20),A);
CALL DISP_COMMON_DATA;
PUT SKIP(2) EDIT ('RECORD SIZE: ',REC_SZ,'BYTES') (R(RFMT));
PUT SKIP(2) EDIT ('FILE SIZE: ',FILE_SZ,'BYTES') (R(RFMT));
PUT SKIP(2) EDIT('OVERFLOW FILE SIZE: ',O_FILE_SZ,'BYTES') (R(RFMT));
PUT SKIP(2) EDIT ('TF (SEARCH ARGUMENT IS NOT THE KEY): ',TF1,'MS')
(R(RFMT));
PUT SKIP(2) EDIT ('TF (SEARCH ARGUMENT IS  THE KEY): ',TF2,'MS')
(R(RFMT));
PUT SKIP(2) EDIT ('TN: ',TN,'MS') (R(RFMT));
PUT SKIP(2) EDIT ('TI: ',TI,'MS') (R(RFMT));
PUT SKIP(2) EDIT ('TU: ',TU,'MS') (R(RFMT));
PUT SKIP(2) EDIT ('TX: ',TX,'MS') (R(RFMT));
PUT SKIP(2) EDIT ('TY: ',TY,'MS') (R(RFMT));
END SEQ_RES_DISP;
END SEQUENTIAL;
/*****************************************************
**       INDEXED SEQUENTIAL CALCULATIONS             **
*****************************************************/
INDEXED_SEQUENTIAL : PROCEDURE;
 DECLARE
 FAN_OUT              FIXED DEC(7),
 NO_OF_LVLS           FIXED DECIMAL(3),
 (O_PF,
  N_F,
 P_OF_OV )            FLOAT DEC(6),
 (SI,
 rec_sz,
 REC_TOT)             FIXED DECIMAL(15,2),
 O_N                  FIXED DEC(10,2);
DECLARE
```

```
 (INDEX,
  TOT_INDEX,
  INDEX_BLKS)          FIXED DECIMAL(8),
  O_PRIME              FIXED DEC(8,2);
DECLARE
  (TF,TFO,TFC,TFM,
   TN,
   TI,
   TU,
   TXSQ,TXSE,
   TY,
   O_FILE_SZ,
   FILE_SZ )           FIXED DEC(15,2);
O_N = NO_INSRT + NO_UPDTE;
FAN_OUT = FLOOR(BLOCK_SZ / (VALUE_SZ + PTR_SZ));
IF FAN_OUT = 0 THEN
  DO;
   FLAG = '5';
   SIGNAL ERROR;
  END;
INDEX = CEIL(N / BFR); /* # OF INDEX ENTRIES IN LEVEL 1 */
INDEX_BLKS = CEIL(INDEX / FAN_OUT); /* SPACE FOR ENTRIES IN LEVEL 1 */
TOT_INDEX = INDEX_BLKS;
DO WHILE( INDEX_BLKS
 INDEX_BLKS = CEIL (INDEX_BLKS / FAN_OUT);
 TOT_INDEX = TOT_INDEX + INDEX_BLKS;
END;
SI = TOT_INDEX * BLOCK_SZ;
REC_SZ = A * VALUE_SZ  + PTR_SZ;
O_FILE_SZ = O_N * REC_SZ;
REC_TOT = REC_SZ + (O_N / N) * REC_SZ + (SI / N);
FILE_SZ = REC_SZ  * N;
NO_OF_LVLS = CEIL(LOG (CEIL(N / BFR)) / LOG (FAN_OUT));
O_PRIME = 0.5 * O_N;
O_PF =  O_N / 2 ;
N_F = N + O_PRIME;
P_OF_OV = O_PF / (N_F);
TFM = S + R + (NO_OF_LVLS -1) * (R + BTT) + R + BTT;
TFO = P_OF_OV * (S + R + BTT);
TFC = 0.5 * BFR * P_OF_OV * P_OF_OV * (R + BTT);
TF = TFM + TFO + TFC;
TN = ((1 - P_OF_OV) / BFR + P_OF_OV) * (R + BTT);
TI = TF + 2 * TRW + R + BTT;
TU = TF + TRW + TI;
TXSE = TF + (N + O_PRIME - 1) * TN;
TXSQ = (N + O_N) * REC_SZ / T_PRIME;
TY = (N + O_PRIME * BFR) / BFR * (R + BTT) +
      (N + O_N - NO_DLTE) * REC_SZ / T_PRIME + (SI / T_PRIME);
CALL INDSEQ_RES_DISP;
RETURN;
/*****************************************************
**       INDEXED SEQUENTIAL RESULTS DISPLAY         **
*****************************************************/
INDSEQ_RES_DISP: PROCEDURE;
RFMT: FORMAT(COL(10),A,COL(55),F(15,2),X(1),A);
```

```
PUT SKIP(2) EDIT ('INDEXED SEQUENTIAL FILE PERFORMANCE PARAMETERS')
   (COL(20),A);
PUT SKIP(0) EDIT ('_____________________________________________')
   (COL(20),A);
CALL DISP_COMMON_DATA;
PUT SKIP(2) EDIT ('RECORD SIZE: ',REC_SZ,'BYTES') (R(RFMT));
PUT SKIP(2) EDIT ('FILE SIZE: ',FILE_SZ,'BYTES') (R(RFMT));
PUT SKIP(2) EDIT ('OVERFLOW FILE SIZE:',O_FILE_SZ,'BYTES') (R(RFMT));
PUT SKIP(2) EDIT ('INDEX FILE SIZE: ',SI,'BYTES') (R(RFMT));
PUT SKIP(2) EDIT ('TF: ',TF,'MS') (R(RFMT));
PUT SKIP(2) EDIT ('TN: ',TN,'MS') (R(RFMT));
PUT SKIP(2) EDIT ('TI: ',TI,'MS') (R(RFMT));
PUT SKIP(2) EDIT ('TU: ',TU,'MS') (R(RFMT));
PUT SKIP(2) EDIT ('TX(SERIAL) ',TXSE,'MS') (R(RFMT));
PUT SKIP(2) EDIT ('TX(SEQUENTIAL) ',TXSQ,'MS') (R(RFMT));
PUT SKIP(2) EDIT ('TY: ',TY,'MS') (R(RFMT));
END INDSEQ_RES_DISP;
END INDEXED_SEQUENTIAL;
/*****************************************************
**       INDEXED FILE CALCULATIONS                  **
*****************************************************/
INDEXED: PROCEDURE;
 declare
   SI           FIXED DECIMAL(15,1),
   SI_TOT       FIXED DECIMAL(15,1),
   NO_OF_LEVEL FIXED DECIMAL(9),
   REC_TOT      FIXED DECIMAL(15,2),
   EF_FANOUT    FIXED DECIMAL(9,2),
   FANOUT       FIXED DECIMAL (7),
   DENS         FIXED DECIMAL (8,7),
   N_PRIME      FIXED DEC (11,2),
   I            FIXED DECIMAL (3),
   PROB_SPLIT    FIXED DECIMAL (3,2),
   T_NEWCOPY     FIXED DECIMAL (13,2),
   T_INDEX       FIXED DECIMAL (13,2),
   T_SPLIT       FIXED DECIMAL (13,2),
   X     FIXED DECIMAL (15,2),
   T_FIXPTR      FIXED DECIMAL (13,2);
 DECLARE (TF,
          TN,
          TI1, TI2, TI,
          TU,
          TXSE, TXSQ,
          TY, TYI,
          FILE_SZ,
          REC_SZ)       FIXED DEC (15,2);
   IF (A_UPD > A) THEN
     DO;
       FLAG = '4';
       SIGNAL ERROR;
     END;
   DENS = DENSITY / 100;
   N_PRIME = N * A_PRIME / A;
   FANOUT = FLOOR (BLOCK_SZ / (VALUE_INDX_SZ + PTR_SZ));
   IF FANOUT = 0 THEN
```

```
     DO;
      FLAG = '6';
      SIGNAL ERROR;
     END;
    EF_FANOUT  = DENS * FANOUT;
    NO_OF_LEVEL = CEIL(LOG (N_PRIME) / LOG(EF_FANOUT));
    SI_TOT = N_PRIME * (VALUE_INDX_SZ + PTR_SZ) / DENS;
    SI = SI_TOT;
    I = 1;
    DO WHILE (I <  NO_OF_LEVEL);
     SI = CEIL(SI / EF_FANOUT);
     SI_TOT = SI_TOT + SI;
     I = I + 1;
    END;
    REC_SZ = A_PRIME * (ATTR_LEN + VALUE_SZ + 2);
    REC_TOT = REC_SZ + SI_TOT / N;
    FILE_SZ = REC_SZ * N + SI_TOT;
     TF = (NO_OF_LEVEL + 1) * (S + R + BTT);
     TN = S + R + BTT;
     PROB_SPLIT = (2 / EF_FANOUT);
     TI1 =  (1 + A_PRIME * (NO_OF_LEVEL + PROB_SPLIT )) *
        (S + R + BTT);
     TI2 = (1 + A_PRIME * (1 + DEC((4 / EF_FANOUT),9,2))) * TRW;
     TI = TI1 + TI2;
     T_NEWCOPY = S + R + BTT + TRW;
     T_INDEX = NO_OF_LEVEL * (S + R + BTT) + TRW;
     T_SPLIT = (S + R + BTT + TRW);
     T_FIXPTR = NO_OF_LEVEL * (S + R + BTT) + TRW;
     TU =  TF + TRW + T_NEWCOPY + 2 * A_UPD * (T_INDEX +
     DEC((PROB_SPLIT * T_SPLIT),9,2)) + (A_PRIME - A_UPD) * T_FIXPTR;
     /* FOLLOWING SERIAL READING */
     TXSE = (TF + (N - 1) * TN) / 3600000;
     /* FOLLOWING SPACE ALLOCATION */
     TXSQ = (CEIL(N / BFR) * (S + R + DEC((BLOCK_SZ / T_PRIME),15,2)))
              / 3600000;
     TYI = TXSQ + (SORT(N_PRIME,EF_FANOUT,BTT) / 3600000)
          + (SI_TOT / T_PRIME) / 3600000;
     TY = (2 * TXSQ ) + A * TYI;
    CALL IND_RES_DISP;
    RETURN;
/*******************************************************
**       INDEXED  RESULTS DISPLAY                     **
*******************************************************/
IND_RES_DISP: PROCEDURE;
RFMT: FORMAT(COL(10),A,COL(55),F(15,2),X(1),A);
CALL DISP_COMMON_DATA;
PUT SKIP(2) EDIT ('INDEXED FILE PERFORMANCE PARAMETERS')
   (COL(20),A);
PUT SKIP(0) EDIT ('_____________________________________')
   (COL(20),A);
PUT SKIP(2) EDIT ('RECORD SIZE: ',REC_SZ,'BYTES') (R(RFMT));
PUT SKIP(2) EDIT ('FILE SIZE: ',FILE_SZ,'BYTES') (R(RFMT));
PUT SKIP(2) EDIT ('INDEX FILE SIZE: ', SI_TOT,'BYTES') (R(RFMT));
PUT SKIP(2) EDIT ('TF: ',TF,'MS') (R(RFMT));
PUT SKIP(2) EDIT ('TN: ',TN,'MS') (R(RFMT));
```

```
PUT SKIP(2) EDIT ('TI: ',TI,'MS') (R(RFMT));
PUT SKIP(2) EDIT ('TU: ',TU,'MS') (R(RFMT));
PUT SKIP(2) EDIT('TX(SERIAL): ',TXSE,'HOURS') (R(RFMT));
PUT SKIP(2) EDIT ('TX(BY BLOCK ALLOCATION)',TXSQ,'HOURS') (R(RFMT));
PUT SKIP(2) EDIT('TY(INDEX ONLY):',TYI,'HOURS') (R(RFMT));
PUT SKIP(2) EDIT ('TY(FILE): ',TY,'HOURS') (R(RFMT));
END IND_RES_DISP;
END INDEXED;
/*****************************************************
**       DIRECT FILE CALCULATIONS                   **
*****************************************************/
DIRECT: PROCEDURE;
DECLARE
 P         FIXED DECIMAL (7,2),
 P1U       FIXED DECIMAL (7,4),
 REC_SZ    FIXED DECIMAL (9,2),
 O_N       FIXED DEC(8),
 CN        FLOAT DEC(6),
 C_PRIME   FLOAT DEC(6);
DECLARE
  (TF,
   TN,
   TI,
   TU,
   TX,
   TLOAD,
   TY,
   O_FILE_SZ,
   FILE_SZ)              FIXED DEC(15,2);
DCL ALPHA   FLOAT DEC(6);
ALPHA = N / M;
IF COLL_TYPE = 1 THEN     /* OPEN ADDRESSING */
DO;
 FILE_SZ = M * A * VALUE_SZ;
 REC_SZ = A *  VALUE_SZ;
 CALL ACCS(2,ALPHA,BFR,CN,C_PRIME);
 P1U = N / M;
 P = CN;
END;
IF COLL_TYPE = 2 THEN    /* SEPERATE CHAINING */
DO;
 P1U = 1 - EXP(N/M);
 O_N = N - M * P1U;
 FILE_SZ = (M + O_N) * (A * VALUE_SZ + PTR_SZ);
 REC_SZ = A *  VALUE_SZ + PTR_SZ;
 O_FILE_SZ = O_N * REC_SZ;
 CALL ACCS(1,ALPHA,BFR,CN,C_PRIME);
 P = CN;
 P1U = C_PRIME;
END;
TF = (S + R + BTT) * P;
TN = TF;
TI = (S + R + BTT + TRW) *  P1U;
TU = TF + TRW;
IF COLL_TYPE = 1 THEN
```

```
  O_N = 0;
TX = (M + O_N) * (R + WASTE) / T_PRIME;
TLOAD = SORT(N,BFR,BTT) + TX;
TY = TX + TLOAD;
CALL DIRECT_RES_DISP;
RETURN;
/*******************************************************
**       DIRECT RESULTS DISPLAY                       **
*******************************************************/
DIRECT_RES_DISP: PROCEDURE;
RFMT: FORMAT(COL(10),A,COL(55),F(15,2),X(1),A);
PUT SKIP(2) EDIT ('DIRECT FILE PERFORMANCE PARAMETERS')
   (COL(20),A);
PUT SKIP(0) EDIT ('_____________________________________')
   (COL(20),A);
CALL DISP_COMMON_DATA;
PUT SKIP(2) EDIT ('RECORD SIZE: ',REC_SZ,'BYTES')(R(RFMT));
PUT SKIP(2) EDIT ('FILE SIZE: ',FILE_SZ,'BYTES') (R(RFMT));
IF COLL_TYPE = 2 THEN
PUT SKIP(2) EDIT ('OVERFLOW FILE SIZE: ',O_FILE_SZ,'BYTES')
(R(RFMT));
PUT SKIP(2) EDIT ('TF: ',TF,'MS') (R(RFMT));
PUT SKIP(2) EDIT ('TN: ',TN,'MS') (R(RFMT));
PUT SKIP(2) EDIT ('TI: ',TI,'MS') (R(RFMT));
PUT SKIP(2) EDIT ('TU: ',TU,'MS') (R(RFMT));
PUT SKIP(2) EDIT ('TX: ',TX,'MS') (R(RFMT));
PUT SKIP(2) EDIT ('TY: ',TY,'MS') (R(RFMT));
END DIRECT_RES_DISP;
END DIRECT;
/*******************************************************
**       CALCULATION OF EXTRA ACCESSES FOR DIRECT FILES  **
*******************************************************/
ACCS: PROCEDURE(CODE,ALPHA,B2,CN,C_PRIME);
 DECLARE B FLOAT DEC(6),B2 FIXED DEC(9,2),
          ALPHA FLOAT DEC(6),
          T     FLOAT DEC(6),
      NEW_T     FLOAT DEC(6),
      OLD_T     FLOAT DEC(6),
          CN    FLOAT DEC(6),
      C_PRIME   FLOAT DEC(6),
          I     FLOAT DEC(6),
          TEMP  FLOAT DEC(6),
          CODE  FIXED DEC(3);
 B = B2;
 IF CODE = 1 THEN
 DO;
   IF B >= 190 THEN
    DO;
     CN = 1;
     C_PRIME =1;
    END;
   ELSE
    DO;
     CALL TERM (ALPHA,B,TEMP);
     CALL TB(ALPHA,B,TEMP,T);
```

```
      CN = 1 + ((1 - 0.5 * B * (1 - ALPHA)) * T)
         + (TEMP / 2 * R(ALPHA ,B));
      C_PRIME = 1 + ALPHA * B * T;
     END;
 END;
 IF CODE = 2 THEN
  DO;
     IF B > 150 THEN
      CN = 1;
     ELSE
     DO;
       CN = 1;
       I = 1;
       NEW_T = 0.0;
       DO UNTIL (ABS(NEW_T - OLD_T) < 0.00001);
         OLD_T = NEW_T;
         CALL TERM(ALPHA,I * B,TEMP);
        CALL TB(ALPHA,I * B,TEMP,NEW_T);
        CN = CN + NEW_T;
        I = I + 1;
       END;
    END;
 END;
 R: PROCEDURE(ALPHA,B) RETURNS (FLOAT DEC(6));
  DECLARE     ALPHA FLOAT DEC(6),
              B     FLOAT DEC(6),
              COUNT FIXED DEC(4),
              OLD_SUM FLOAT DEC(6),
              NEW_SUM FLOAT DEC(6),
              TERM    FLOAT DEC(6);
  OLD_SUM = 0;
  COUNT = B + 1;
  NEW_SUM = B / COUNT;
  TERM = NEW_SUM;
  DO WHILE((NEW_SUM - OLD_SUM) > 0.001);
    OLD_SUM = NEW_SUM;
    COUNT = COUNT + 1;
    TERM = TERM * B * ALPHA / COUNT;
    NEW_SUM = OLD_SUM + TERM;
  END;
  RETURN (NEW_SUM);
  END R;
TERM: PROCEDURE (ALPHA,B,TEMP);
   DECLARE ALPHA   FLOAT DEC(6),
           B       FLOAT DEC (6),
           TEMP    FLOAT DEC(6),
           I       FLOAT DEC(6);
/* CALCULATE TEMP = (EXP(-B* ALPHA) * (B**B) * (ALPHA**B) / FACT(B) */
 TEMP = EXP(-B * ALPHA);
 DO I = 1 TO B;      /* TEMP = TEMP * (B**B) */
  TEMP = TEMP * B / I;
 END;
 TEMP = TEMP * (ALPHA ** B);
 END TERM;
TB: PROCEDURE (ALPHA,B,TEMP,T);
```

```
  DECLARE B FLOAT DEC(6),
          ALPHA FLOAT DEC(6),
          T     FLOAT DEC(6),
          TEMP  FLOAT DEC(6);
  T = TEMP * (1 - (1 - ALPHA) * R(ALPHA,B));
 END TB;
 END ACCS;
 END CALC;
```

Contents of DISK PARM file

#	seek	latency	gap	t rate(t)	device
1	16.00	8.30	524.00	3000.00	IBM3380
2	60.00	12.50	200.00	312.00	IBM2319
3	141.00	83.30	60.00	62.00	IBM 851 8" FLOPPY
4	90000.0	250.00	600.00	120.00	IBM 9 TRACK MAGNETIC TAPE

APPENDIX D

RESULTS

PILE FILE INPUT DATA

BLOCK SIZE:	4,096 BYTES
TOTAL NUMBER OF RECORDS IN THE MAIN FILE:	100,000 RECORDS
RECORD TYPE:	VARYING LENGTH RECORDS
POINTER SIZE:	4 BYTES
AVERAGE # OF ATTRIBUTES PER RECORD:	8
AVERAGE ATTRIBUTE_NAME LENGTH:	11 BYTES
AVERAGE ATTRIBUTE_VALUE LENGTH:	13 BYTES
AVERAGE NUMBER OF INSERTS BETWEEN REORGANIZATION	5,000
AVERAGE NUMBER OF DELETES BETWEEN REORGANIZATION	5,000
AVERAGE NUMBER OF UPDATES BETWEEN REORGANIZATION	5,000

PILE FILE PERFORMANCE PARAMETERS

BLOCKING FACTOR:	18.00 RECORDS
BLOCK TRANSFER RATE (BTT):	1.36 MS
BULK TRANSFER RATE (T PRIME):	1263.77 CHAR/MS
RECORD SIZE:	208.00 BYTES
FILE SIZE:	20800000.00 BYTES
TF	8229.34 MS
TN	8229.34 MS
TI	42.26 MS
TU	8288.20 MS
TX (SEQUENTIAL READ)	16458.68 MS
TX (SERIAL READ)	228031.16 MS
TY	34563.25 MS

SEQUENTIAL FILE INPUT DATA

BLOCK SIZE:	3,072	BYTES
TOTAL NUMBER OF RECORDS IN THE MAIN FILE:	10,000	RECORDS
# OF ATTRIBUTES PER RECORD	5	
AVERAGE ATTRIBUTE_VALUE LENGTH:	20	BYTES
AVERAGE NUMBER OF INSERTS BETWEEN REORGANIZATION	100	
AVERAGE NUMBER OF DELETES BETWEEN REORGANIZATION	100	
AVERAGE NUMBER OF UPDATES BETWEEN REORGANIZATION	150	

SEQUENTIAL FILE PERFORMANCE PARAMETERS

BLOCKING FACTOR:	30.00	RECORDS
BLOCK TRANSFER RATE (BTT):	1.02	MS
BULK TRANSFER RATE (T PRIME):	1277.03	CHAR/MS
RECORD SIZE:	100.00	BYTES
FILE SIZE:	1000000.00	BYTES
OVERFLOW FILE SIZE:	50000.00	BYTES
TF (SEARCH ARGUMENT IS NOT THE KEY):	411.10	MS
TF (SEARCH ARGUMENT IS THE KEY):	231.77	MS
TN:	0.07	MS
TI:	44.92	MS
TU:	456.02	MS
TX:	1030.08	MS
TY:	1813.08	MS

INDEXED SEQUENTIAL FILE INPUT DATA

BLOCK SIZE:	1,024 BYTES
TOTAL NUMBER OF RECORDS IN THE MAIN FILE:	10,000 RECORDS
# OF ATTRIBUTES PER RECORD	5
AVERAGE ATTRIBUTE_VALUE LENGTH:	20 BYTES
POINTER SIZE:	4 BYTES
AVERAGE NUMBER OF INSERTS BETWEEN REORGANIZATION	100
AVERAGE NUMBER OF DELETES BETWEEN REORGANIZATION	100
AVERAGE NUMBER OF UPDATES BETWEEN REORGANIZATION	150

INDEXED SEQUENTIAL FILE PERFORMANCE PARAMETERS

BLOCKING FACTOR:	10.00 RECORDS
BLOCK TRANSFER RATE (BTT):	0.34 MS
BULK TRANSFER RATE (T PRIME):	984.25 CHAR/MS
RECORD SIZE:	104.00 BYTES
FILE SIZE:	1040000.00 BYTES
OVERFLOW FILE SIZE:	26000.00 BYTES
INDEX FILE SIZE:	25600.00 BYTES
TF:	41.88 MS
TN:	0.95 MS
TI:	83.72 MS
TU:	142.20 MS
TX(SERIAL)	9659.68 MS
TX(SEQUENTIAL)	1083.05 MS
TY:	10818.49 MS

INDEXED FILE INPUT DATA

BLOCK SIZE:	4,096	BYTES
TOTAL NUMBER OF RECORDS IN THE MAIN FILE:	100,000	RECORDS
RECORD TYPE:	VARYING	LENGTH RECORDS
POINTER SIZE:	4	BYTES
AVERAGE # OF ATTRIBUTES PER RECORD:	8	
TOTAL # OF ATTRIBUTES IN THE FILE:	11	
AVERAGE ATTRIBUTE_NAME LENGTH:	10	BYTES
AVERAGE ATTRIBUTE_VALUE LENGTH:	13	BYTES
AVERAGE LENGTH OF INDEX ATTRIBUTE:	13	BYTES
AVERAGE # OF UPDATE ATTRIBUTES:	5	
DENSITY OF INDEX FILE:	69	%

BLOCKING FACTOR:	19.00	RECORDS
BLOCK TRANSFER RATE (BTT):	1.36	MS
BULK TRANSFER RATE (T PRIME):	1266.67	CHAR/MS

INDEXED FILE PERFORMANCE PARAMETERS

RECORD SIZE:	200.00	BYTES
FILE SIZE:	21801870.00	BYTES
INDEX FILE SIZE:	1801870.00	BYTES
TF:	102.64	MS
TN:	25.66	MS
TI:	795.60	MS
TU:	1382.24	MS
TX(SERIAL):	0.71	HOURS
TX(BY BLOCK ALLOCATION)	0.04	HOURS
TY(INDEX ONLY):	0.04	HOURS
TY(FILE):	0.52	HOURS

DIRECT FILE INPUT DATA

BUCKET SIZE:	1,024 BYTES
TOTAL NUMBER OF RECORDS IN THE MAIN FILE:	10,000 RECORDS
TOTAL NUMBER OF SLOTS FOR THE RECORDS IN THE MAIN FILE:	12,000 RECORDS
COLLISION RESOLUTION TYPE:	CHAINED ACCESS
# OF ATTRIBUTES PER RECORD:	7
AVERAGE ATTRIBUTE_VALUE LENGTH:	20 BYTES
POINTER SIZE:	5 BYTES

DIRECT FILE PERFORMANCE PARAMETERS

BLOCKING FACTOR:	7.00 RECORDS
BLOCK TRANSFER RATE (BTT):	0.34 MS
BULK TRANSFER RATE (T PRIME):	989.31 CHAR/MS
RECORD SIZE:	145.00 BYTES
FILE SIZE:	5453305.00 BYTES
OVERFLOW FILE SIZE:	3713305.00 BYTES
TF:	28.82 MS
TN:	28.82 MS
TI:	62.13 MS
TU:	45.42 MS
TX:	3160.97 MS
TY:	17982.57 MS