Doubly-chained tree data base organisation—analysis and design strategies

Alfonso F. Cárdenas* and James P. Sagamang†

Doubly-chained tree data base (file) structures are analysed and modelled, taking into account realistic implementation-oriented factors. Formulations for average access time and storage requirements are derived, showing the effect on performance and the interrelation of the characteristics of the contents of the data base, the complexity of the queries, storage device characteristics and processing times, and implementation-oriented (e.g. storage mapping) alternatives. Performance results are obtained from these models for six real life data bases. The doubly-chained alternatives are compared and evaluated to provide practical design guidelines and strategies.

(Received March 1975)

1. Introduction

Data base architecturing and implementation is largely performed in an ad hoc and unsystematic manner in actual practice. Useful guidelines and quantitative approaches are needed to improve the state of the art. The subject matter presented here is part of this needed technology.

The focus is on analysing, modelling and evaluating doubly-chained data base (file) organisations. Doubly-chained techniques and other secondary file organisations, such as inverted files, have emerged and evolved in an effort to meet the increasing querying (information retrieval) and performance demands of modern data base users. Sussenguth (1963) suggested double-chained tree structures for file searching and updating. Other authors have also treated tree structures, or concepts, in various forms and with various orientations, giving rise to a number of synonyms and related terms such as 'threaded trees,' 'triply-linked trees' and 'hierarchies.' (Knuth, 1969; Patt, 1969; Lefkovitz, 1969; Dodd, 1969; Stanfel, 1970; Hu, 1972; Casey, 1973; Salton, 1968). It is unfortunate that among the many professionals concerned with data management, e.g. computer scientists, library scientists, data processing specialists, and linguists, lack of consistent and widely accepted terminology, and often even concepts, prevails in general.

Quantitative measures of average access time and storage requirements for the doubly-chained tree data base organisation are derived. These are important first order data base design parameters. The models examined take into account four elements affecting performance: the characteristics of the contents of the data base, the complexity of queries, average storage device characteristics and processing times, and implementation-oriented issues.

Past publications on doubly-chained structures have not analysed formally and integrally all four elements. They have essentially disregarded or treated only lightly query complexity and actual implementation-oriented aspects. General implementation-oriented aspects, as well as specific implementation-dependent details, may have a large effect on data base performance—a fact often underestimated (Cardenas, 1973). The analysis presented considers important generally applicable implementation-oriented issues, rather than microscopic implementation-dependent details.

The 'logical-physical' techniques and results in this work are applicable and relevant whether the data base is viewed logically as a collection of COBOL-like records, or as a collection of tabular entries or tuples of a relation, i.e. as a relational data base (Codd, 1970). In Section 2 the basic concepts and terminology for the doubly-chained tree data organisation are briefly clarified. The basic considerations of data base organisation performance and evaluation are made. Sections 3 and 4 are devoted to the detailed analysis and modelling of three alternative doubly-chained strategies. Expressions for the average access time and storage requirements are derived. Section 5 shows performance results from these models for six real life data bases. The doubly-chained alternatives are compared and evaluated to provide practical guidelines to architecture doubly-chained data base systems. Section 6 concludes this work.

2. Basic considerations

The basic doubly-chained tree organisation is illustrated in **Fig. 1.** The arrows represent pointers. Sussenguth presented it in 1963. Subsequently, such a data organisation approach has been treated in a variety of forms and environments, resulting in synonyms or terms for related data structures such as 'threaded trees' (with and without the dashed pointers), 'triply linked' organisations (Knuth, 1969) and even 'hierarchic' structures (Dodd, 1969).

In Fig. 1 each level represents a keyname (or domain of a relation) and each node represents a keyvalue. The leaves of the tree are the data records in the data base. Three pointers are usually associated with each node. The *F* pointer (*F-PTR*) points to a set of keyvalues on the next lower level which are in those records having the keyvalue denoted by the node. This set of keyvalues is usually called a filial set. The *C* pointer (*C-PTR*) points horizontally to the next keyvalue in the filial set. It can also be used to traverse horizontally from filial set to filial set—in the special case that it is in the last node of a filial set. The *P* pointer (*P-PTR*) points to the keyvalue on the level above which is the parent of the filial set of which the node is a member. These intricacies are best clarified by the self-explanatory example in **Figs. 2 and 3.** In Fig. 3, *A* and *E* form the filial set of *M*; *B*, *C* and *D* form the filial set of *A*; *F* is the filial set of *E*.

Physically, the organisation contains two types of blocks: (*a*) index blocks, containing access keyvalues and associated pointers, and (*b*) data blocks, containing the records without these keyvalues. The index blocks contain the access keyvalues, and associated pointers, selected to enhance the speed of accessing of records. If all keyvalues are placed in the index, the record becomes just an address. The pointers are sufficient for reconstructing the original record from any point in the tree.

When all keyvalues are placed in the index, the whole data

*Computer Science Department, School of Engineering and Applied Science, University of California, Los Angeles, Los Angeles, California 90024, USA. Portions of this work done while on leave at the IBM Research Laboratory, San Jose, California, USA.
†US Navy Weapons Centre, Department of the Navy, Corona, California 91720, USA.

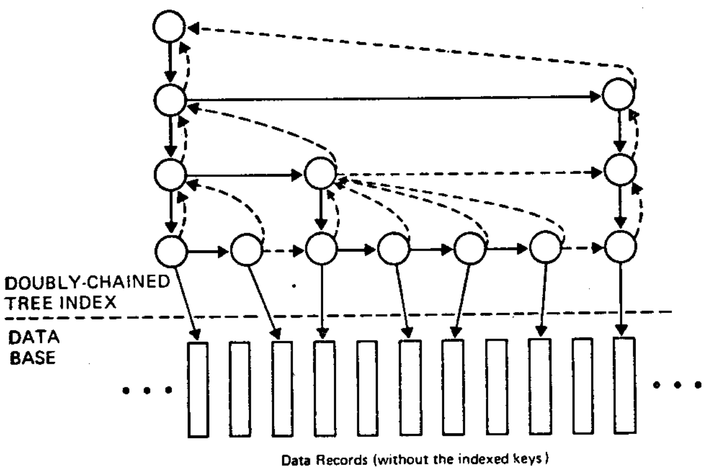


Fig. 1 Basic doubly-chained tree file organisation

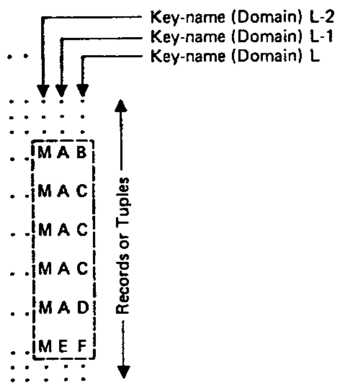


Fig. 2 Values of a subset of (a) Key-names of a set of records or (b) Domains of tuples of a relation

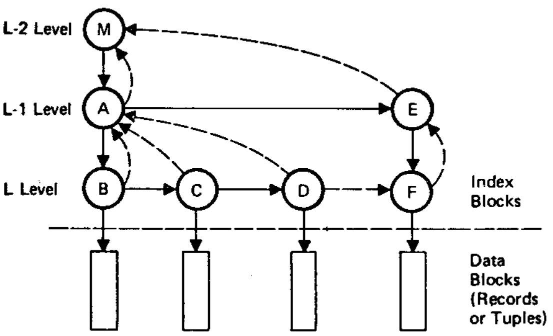


Fig. 3 Doubly-chained tree organisation for the data in Fig. 2.

base is doubly-chained, or in tree form if the backward pointers are omitted. The situation corresponds conceptually to a fully inverted file organisation, except that in the case of inversion keyvalues remain in the record since no pointers are available for reconstructing the original record. Tree representations of data have of course been in use in numerous applications for a long time. However, the use of the doubly-chained approach as a secondary file organisation in a manner analogous to inversion has not been so clearly envisioned or exercised.

Keys which appear often as access keys in a query are the ones which should be indexed. The doubly-chained conglomeration of these keys shall be called doubly-chained tree index or directory. The same general considerations that apply to the selection of keys for inversion in inverted file organisations apply to the selection of keys for double-chaining. However, the structure and hence access time and storage behaviour of these organisations differ. Although the issue of what keys to index is not the main focus in this work, the modelling and determination of access time is essential towards this end. King (1974) has suggested an approach to index selection for inverted organisations which could be adapted and coupled with the analysis here to determine the degree of double-chaining.

It is obvious that if keynames appear in a query which are not indexed, then the whole data record space has to be searched if a logical OR operator (a disjunction) appears in the query. But if only AND operators (conjunctions) appear, then only the subset of the data record space covered by the indexed keys need to be searched.

A significant advantage of the doubly-chained organisation is that it requires less storage space than the original sequential file or any other data base organisation. This results from the

Table 1 Parameters and symbols used

Query transaction characteristics:	
*ACI**	Average number of atomic conditions per item condition
*ICR**	Average number of item conditions per record condition
*RCQ**	Average number of record conditions per query condition
RAVE	Average number of records retrieved per query, past statistics
NKEY	Number of access key-names or levels in the doubly-chained tree
Data base statistics:	
NREC	Number of records in the data base
RLAVE	Average record length
KLAVE	Average key-value length
NVAL(I)	Number of unique key-values for the *I*th key
DKV	Sum of the number of distinct key-values for the *NKEY* key-names
NODE(I)	Number of nodes on the *I*th level of the doubly-chained tree
Parameters estimated:	
KLFIX	Fixed key-value length
BFI	Blocking factor (index block)
RSI	Reserve space (index block)
SRI	Storage requirement (index blocks)
BFD	Blocking factor (data block)
RSD	Reserve space (data block)
SRD	Storage requirement (data blocks)
LSTAVE	Average list length for the *NKEY* keys
X_N	Average number of nodes processed or examined per query
X_I	Average number of index blocks accessed per query
X_D	Average number of data blocks accessed per query
TOTSR	Total storage requirement
ACCTM	Average access time

*see Appendix 1

Table 2 Device parameters and symbols used

BLOCKC	Block length in characters, 10,740 characters*
BLOCKW	Block length in words, 1,790 words*
WORD	Word length in characters, 6 characters*
T_T	Average time to access a block, track or page, 100 msec.*
T_N	Average time to examine or process a node, 1.5 msec.*

*Parameter values based on the UNIVAC FASTRAND drum storage device and estimated average processing times removal of the indexed keys from the data base records and of redundant values within filial sets. Storage savings for actual data bases will be illustrated in Section 5. A disadvantage is that programming may be complicated if sound implementation approaches are not followed.

The performance of any file organisation is a function of (Cardenas, 1973):

1. Data base characteristics, such as the distribution of values of access keys (access domains);
2. User requirements, particularly the complexity of the queries:
3. Storage device and processing time specifications, such as the average time to access a track on a disc;
4. The particular data base organisation, and the general programming and storage approaches towards its implementation, i.e. implementation-oriented factors;
5. The specific implementation-dependent aspects, such as particular programming aspects, the actual encoding and mapping of data items and pointers on storage, etc.

The analysis presented here takes into account factors 1, 2, 3 and 4.

It is important to distinguish between implementation-oriented aspects and implementation-dependent details. It is a fact that there are numerous options and intricate data and file handling issues and peculiarities that characterise storage devices and the supporting operating system basic data management. They unfortunately differ from device to device and operating system to operating system (differences which cause costly lack of data base transferability, reprogramming efforts, etc.). For example, overflow management in the indexed sequential organisation differs from system to system. While it is acknowledged that actual implementation details may have underestimated effects on performance (Cardenas, 1973), it would be extremely difficult and possibly not cost effective to account for such microscopic and implementation dependencies in modelling and deriving access time and storage requirements. But is is necessary to consider first order and generally applicable implementation-oriented aspects.

While at the conceptual logical-physical level the doubly-chained organisation is well defined, as illustrated in Fig. 2 and 3, at a lower and implementation-oriented level there are alternative doubly-chained search strategies and physical structures. These alternatives and issues are elucidated in the following sections, as the expressions for average access time and storage requirements are derived. In Section 3 two alternative doubly-chained strategies for the same physical organisation and layout are modelled. In Section 4 a different physical organisation and layout is analysed. These alternatives or versions as they will be called, are quantitatively compared and evaluated in Section 5. Although the versions presented are the ones considered to be most appropriate and to cover the span of possibilities, it is not claimed that they are the only possible strategies.

Tables 1 and **2** define the parameters and symbols used. Appendix 1 defines the manner in which queries and their complexity are characterised here. Practically all types of queries can be placed into the format indicated. Unnecessary mathematism is avoided where justified to facilitate understanding.

The following integer rounding convention is used. If X is a number, it can be represented by $X = INT + R$ where INT is an integer and R is a fractional remainder such that:

$$[X]_- = INT$$
$$[X]_+ = INT \text{ if } R = 0$$
$$\quad\quad INT + 1 \text{ if } R \neq 0$$
$$[X] \;\; = \text{rounding}$$

Fixedlength keyvalues are assumed. However, the variable length situation is accommodated easily in the results obtained by allowing for an extra 'length' field in every node.

It is assumed realistically that the data base and the node-pointer space are mapped on some secondary storage space which is eventually segmented into blocks or tracks or pages. The basic file accessing mechanisms, e.g. at the COBOL or PL/I level, underlying a data base system are block-oriented in the sense that the tendency is for blocks to be accessed almost as easily as a single record in the block. As an example, the three versions of the random access file organisation supported in PL/I are cited: Regional (1), Regional (2) and Regional (3) (IBM, 1975; Weinberg, 1970); random accessing is to regions (part of a track or a complete track), which contain from one to several records (the 'randomly' organised records within a track are really searched sequentially). Thus the basic unit of data I/O used here is the block transfer, **Table 2.**

A node is treated as a record made up of the keyvalue and pointers, with a unique address in the node data file. If the number of nodes is very large, then the node data file becomes a serious data base problem in itself. This realisation is reflected in the analysis presented.

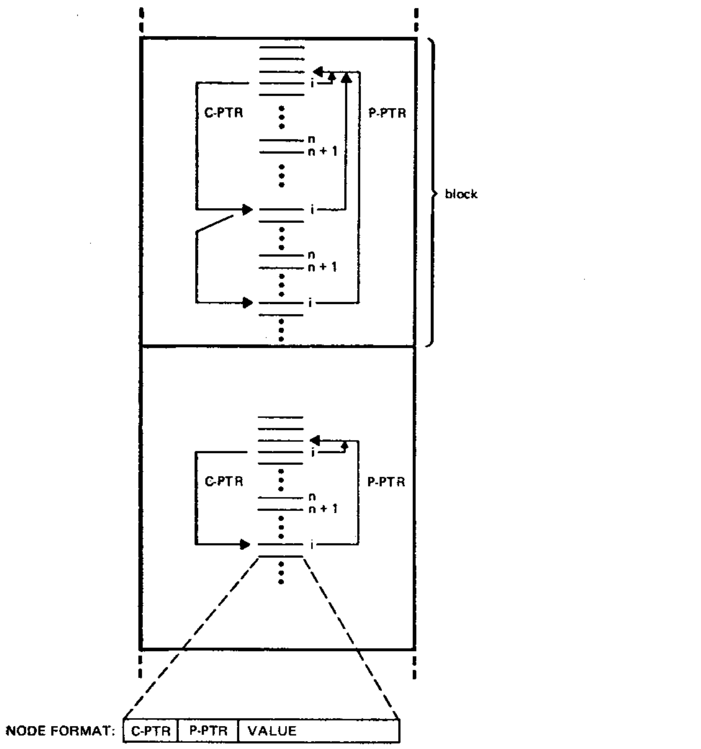
3. Doubly-chained structure versions 1 and 2

The physical layout envisioned for versions 1 and 2 is shown in **Fig. 4.** These versions use implied *F-PTR*'s. The *F-PTR* is implicitly used by the fact that the first member of the filial set of a node is physically stored immediately following that node. In the basic structure shown in Fig. 1, the *F-PTR* in the nodes on the last level of the tree pointed to data records. In these versions, data records are referenced by a special node. Its VALUE is a pointer to an 'overflow' data block; its *C-PTR* points to the first and its *P-PTR* points to the last of a string of consecutive data records. Assuming that n accesskeys have been specified, this special node is labelled $n + 1$ in Fig. 4.

With the exception of the special node, pointers may not cross block boundaries. This is illustrated in Fig. 4. This may be difficult to accomplish completely or in all blocks in cases of tree levels with long filial sets. The *C-PTR* and *P-PTR* links are shown for a filial set on level i. Logically they still represent a single filial set, but physically they are implemented as two independent filial sets. Another helpful way of viewing this is that each index block can be thought of as representing a complete file. The example in Figs. 2, 3 and **Fig. 5** illustrates and clarifies the structuring.

This strategy provides good response for updates. This is a qualitative judgment and is best justified by giving a brief description of the file generation and update strategy. To generate or update this organisation, the input records must be sorted according to the doubly-chained keynames. The file generation algorithm sequentially reads each record of the data base and sequentially processes the keyvalues. Assuming n keynames have been defined, the algorithm saves $n + 1$ pointers which point to the last nodes processed and a pointer to the next available location. Given a new record the algorithm compares the access keyvalues to the corresponding nodevalues. If all are equal the datarecord is stored and the *P-PTR* in the special node is incremented. If some comparison is not equal, say at level i, then the new value is stored in the next available position, the necessary linkages made, and the saved pointer for level i is reset. The values corresponding to level $i + 1$, $i + 2$, $i + 3$, . . ., n are then stored sequentially, the pointers set, the datarecord stored, and the special node is created. In the event that a comparison is not equal and overflow is indicated, the block is written out. The values from the current record are then stored in the first n positions and the algorithm proceeds as if this were the first record in the file.

Now, if a new record is to be added to an existing file and it reflects a new value at level i, a search is performed on filial sets



NOTE: The F-PTR is implicit since the first member of the filial set of a node is physically stored immediately following that node.

Fig. 4 Doubly-chained tree organisation, versions 1 and 2

Relative Address	Node C–PTR	Node P–PTR	Value	
⋮	⋮	⋮	⋮	
105	–	–	M	
106	113	105	A	
107	109	106	B	
108	R1001	R1001	overflow pointer	
109	111	106	C	Filial Set of A; Filial Set of M
110	R1002	R1004	overflow pointer	
111	114	106	D	
112	R1005	R1005	overflow pointer	
113	–	105	E	
114	–	113	F	Filial Set of E
115	R1006	R1006	overflow pointer	
116	⋮	⋮		

NOTE: Each record or tuple (without the indexed keys) is stored in relative address Rxxxx.

Fig. 5 Actual physical structure corresponding to the doubly-chained organisation, versions 1 and 2, Fig. 4 for the data in Fig. 2.

for levels 1, 2, 3, . . ., *i*, moving from block to block until the correct place for the new value is located. If the key-value is greater than the nodevalue, the keyvalues are inserted sequentially as before except now they go in the reserve area. If, on the other hand, the keyvalue is less than the nodevalue one will have to back up to the previous block. Thus, it is necessary to back up only in this special case. If pointers had been arbitrarily allowed to cross block boundaries, a great deal of jumping back and forth would be required.

The equations for storage requirements will now be derived. The first step is to specify the fixed length, *KLFIX*, for the node values. This will be the average key length, *KLAVE*, in terms of the word unit of the particular computer, expanded to a full word boundary:

$$KLFIX = \left[\frac{KLAVE}{WORD}\right]_+ \quad (1)$$

where *WORD* is the length of a word in characters. Allowing one word each for *C-PTR* and *P-PTR*, the blocking factor, *BFI*, for the index blocks is given by:

$$BFI = \left[\frac{BLOCKW}{KLFIX + 2}\right]_- \text{ nodes/block} \quad (2)$$

Assuming a 10 per cent reserve space, *RSI*, for index blocks:

$$RSI = \left[\frac{BFI}{10}\right]_+ \text{ reserve node space/block} \quad (3)$$

The choice of a 10 per cent reserve area is for the purpose of simplification. Some applications may require no reserve space. Others may require more than 10 per cent or more elaborate reserve space management. This is determined from a consideration of the frequency and type of updating (insertion, deletion) for the specific application.

The number of nodes, $NODE(I)$, on each level of the tree is obtained from data base measurements. The notation $SUMM(NODE(I))$ will be used to indicate a summation over all defined tree levels, $\sum_{I=1}^{NKEY} NODE(I)$. The number of tree levels is the number of indexed access key names, *NKEY*, used in the queries. Thus, the number of nodes at the lowest level is given by $NODE(NKEY)$. For each node on this level, a special node has been added. Based on this, the storage requirement, *SRI*, in blocks is given by:

$$SRI = \left[\frac{NODE(NKEY) + SUMM(NODE(I))}{BFI - RSI}\right]_+ \text{ blocks} \quad (4)$$

Whereas the number of objects per block for the index blocks is a definite number, the data blocks have a variable number of objects. The blocking factor, therefore, is the average number of data records per block. Since key values are not stored in the data blocks, the length of data records must be computed from the average (input) record length, *RLAVE*, and the average key length, *KLAVE*, obtained via measurements. Thus, the blocking factor, *BFD*, and the reserve space, *RSD* of the data blocks are:

$$BFD = \left[\frac{BLOCKC}{RLAVE - (NKEY*KLAVE)}\right]_- \text{ records/block} \quad (5)$$

$$RSD = \left[\frac{BFD}{10}\right]_+ \text{ records/block} \quad (6)$$

The determination of storage requirement for the data blocks, *SRD*, uses the actual number of records, *NREC*, in the data base

$$SRD = \left[\frac{NREC}{BFD - RSD}\right]_+ \text{ blocks} \quad (7)$$

Thus, the total storage required *TOTSR* is:

$$TOTSR = SRI + SRD \text{ blocks} . \quad (8)$$

To compute average access time, two basic units of time must be defined:

1. T_T—the average time to access a track including head positioning, latency, and transmission rate.
2. T_N—the average time to process a node in the tree, that is, to examine its *VALUE* and/or traverse it.

Based on these units of time, the average access time is a function of:

1. The number of index blocks that must be accessed, X_I.
2. The number of data blocks that must be accessed, X_D.
3. The number of nodes that must be processed, X_N.

Expressed in these terms the average access time *ACCTM* is

$$ACCTM = (T_N * X_N) + T_T(X_I + X_D) \text{ seconds} . \quad (9)$$

For discussing search strategies in the tree structure, it is convenient to think in terms of filial sets. A commonly occurring characteristic is the size of a filial set. The average filial set size, S, at level I of the tree will be defined as (a local definition for the size of filial sets is taken):

$$S(I) = \left[\frac{NODE(I)}{NODE(I-1)}\right]_+ \quad I = 1, 2, 3, \ldots NKEY \qquad (10)$$

where $NODE(0)$ is defined to be one.

It is assumed in various subsequent formulations that the key values of keys are or tend to be uniformly distributed. Thus the averages used. However, non-uniform key value distributions could be accommodated with the corresponding treatment.

Before describing the search strategies, the following probabilistic considerations must be made to determine the number of data blocks X_D that will be accessed. There are N records distributed among M blocks; there is a fixed number of records per block, N/M. Suppose that K records are to be retrieved. What is the probability that they are in $K, K-1, K-2, \ldots, 1$ blocks? The expression $M(1-(1-1/M)^K)$ gives the average number of blocks that contain the K records and that will have to be retrieved (its derivation may be an interesting exercise for the reader). M is the variable SRD, equation (7). As $M \rightarrow \infty$, the number of blocks retrieved will approach K. As $K \rightarrow \infty$, the number of blocks retrieved will approach M. It is possible that K may be much greater than the total number of data blocks; in this situation SRD is the upper bound of blocks accessed, since K records tend to be in SRD blocks, assuming uniform distribution of K over the data blocks (no clustering). The value of K will be directly proportional to the average list length $LSTAVE$:

$$LSTAVE = \frac{NREC}{\frac{1}{NKEY}\sum_{I=1}^{NKEY} NVAL(I)} = \frac{NKEY * NREC}{SUMM(NVAL(I))} \qquad (11)$$

$LSTAVE$ is the average number of records that are characterized by a unique access key, assuming that the key values of the key names are uniformly distributed. Furthermore, the number of records K that will be referenced by a query will depend on the number of access keys and logical operations used in the query. Thus, $K = ACI * RCQ * LSTAVE$. ACI and RCQ characterise query complexity as defined in Appendix 1. Note that $SUMM(NVAL(I))$ is the number of distinct key values DKV in the $NKEY$ key names, and that $SUMM(NVAL(I)) \leqslant TNODES$ where $TNODES$ is the total number of nodes in the index.

Two possible search algorithms can be specified depending on whether or not the K key names used in the characteristic query correspond to the first K levels in the doubly-chained structure. If the structure is such that there is no such match, it will be called Version 1; if there is, it will be termed version 2. The following paragraphs analyse first version 2; version 1 is then treated.

Version 2

Version 2 assumes that the characteristic query has an item condition (i.e. key name R value$_1$ OR key name R value$_2$ OR . . ., where R denotes one of the set $>$, $<$, $=$) for each of K key names which corresponds to the first K levels in the tree. Refer to Appendix 1 for a definition of query characteristics. First search level 1. When a condition is satisfied at some node, save that node address and move to its filial set. Repeat the process until K conditions have been satisfied. Then move to the $NKEY$th level and obtain pointers to the first and last record satisfying the record condition. The search is resumed in the filial set on the Kth level. When a condition can no longer be satisfied, the search is resumed on the $(K-1)$st level by following the P-PTR. Continue in this manner until level 1 is reached. It is important to note that, because of the physical structuring technique chosen, the node on the level above will be in the current block, although when the number of levels and size of the filial sets is large the node on the previous level above may be in a preceding block. When level 1 is reached, the search proceeds as at the start.

Since filial sets are ordered, the average number of comparisons for a single atomic condition (see Appendix 1) is $S(I)/2$. Hence, the equation for average number of nodes processed X_N is

$$X_N = ACI * \sum_{I=1}^{ICR} \frac{S(I)}{2} \text{ where } S(I) = \frac{NODE(I)}{NODE(I-1)} \qquad (12)$$

where ICR (see Appendix 1) represents the average value of K. The expression for X_N is best justified by considering the existence of a procedure for searching a filial set. The procedure is given the address of the first node and a value for comparison. The filial set is actually an ordered set of values. The procedure returns an address of a node and a 'found' indicator. If found, the address is for the node that was equal to the input value. If not found, the address is for the node that has a value just smaller than the input value. This procedure is called for each atomic condition. The average number of comparisons made by the procedure is $N/2$, where N is the number of nodes in the filial set. The average number of nodes in a filial set may vary considerably from level to level; so the average size for each level is used. Thus, the average number of nodes processed at each Ith level is $ACI * S(I)/2$. This has to be done for each level indicated by the record condition ICR.

$$X_N = \sum_{I=1}^{ICR} ACI * \frac{S(I)}{2} = ACI * \sum_{I=1}^{ICR} \frac{S(I)}{2} . \qquad (13)$$

It should be noted that if the query includes 'less than X' or 'greater than X' qualification operators, not just 'equal to X' operators, the previous and subsequent formulations still hold under the assumption that the distribution of occurrences of 'less than X' and 'greater than X' will average to having the same effect as 'equal to X'. If a 'not equal to X' qualification appears, then exhaustive sequential searches are required at the key level of X. The formulations herein could be modified accordingly to accommodate this possibility. These are obviously areas of further subsequent research.

If it is assumed that the key values on level I are uniformly distributed among the index blocks, then

$$X_I = \frac{SRI}{2} . \qquad (14)$$

if and only if the number of levels $NKEY$ in the index is equal to ICR, and not greater. This gross approximation invites further refinement.

The average number of data blocks is

$$X_D = \min\left[\frac{NREC}{NODE(ICR)}, SRD * (1-(1-1/SRD)^K)\right] \qquad (15)$$

where $K = ACI * LSTAVE$.

The term $NREC/NODE(ICR)$ is a more refined estimate for the specific situation for which these statistics are gathered.

The search strategy and equations above have been given for a single record condition, i.e. $RCQ = 1$ (see Appendix 1). If query conditions are to be used, the search strategy remains essentially the same. All of the different record conditions can be resolved during the same pass as long as the record conditions are kept separate. The equations, however, must be modified:

$$X_N = RCQ * ACI * \sum_{I=1}^{ICR} \frac{S(I)}{2} \qquad (16)$$

$$X_I = \frac{SRI}{2} \tag{17}$$

$$X_D = \min\left[\frac{RCQ * NREC}{NODE(ICR)}, SRD(1 - (1 - 1/SRD)^K)\right]$$

$$\text{where } K = ACI * RCQ * LSTAVE\ . \tag{18}$$

Version 1

Version 1 will now be analysed. This strategy is for environments in which data base querying is on the basis of any combination of K access keys, assuming that K is an arbitrary subset of the $NKEY$ keys. This situation simplifies the analytic modelling process. How this affects the search strategy is best seen in terms of an atomic condition. All key values for a given key name are stored in some level of the tree. A given level of the tree is divided into filial sets. Within a filial set key values are unique, but between filial sets values may be repeated. Thus, given an atomic condition for some arbitrary level of the tree, *all* filial sets must be searched on that level. For this reason, all index blocks must be accessed:

$$X_I = SRI\ . \tag{19}$$

In this case, the compution of X_D will be based only on average list length, $LSTAVE$, equation (11). The determination of X_N and X_D must be given on the basis of the logical operations which will be used in queries. This is indicated below in terms of query complexity as defined in Appendix 1.

1. Atomic condition ($ACI = 1$). The average number of comparisons for a filial set at level I in the tree is $S(I)/2$. There are $NODE(I - 1)$ filial sets on level I. Averaging this over all levels gives the value of X_N.

$$X_N = \frac{1}{2 * NKEY} SUMM(NODE(I - 1) * S(I)) \tag{20}$$

$$X_D = SRD * (1 - (1 - 1/SRD)^K) \tag{21}$$

$$\text{where } K = LSTAVE\ .$$

2. Item condition ($ACI \geq 2$). An item condition is a disjunction of atomic conditions for the same key name. As a filial set is searched, a test is made for each atomic condition. Thus, the average number of comparisons is $ACI * (S(I)/2)$. Extending this to a level and tree search:

$$X_N = \frac{ACI}{2 * NKEY} SUMM(NODE(I - 1) * S(I)) \tag{22}$$

$$K_D = SRD(1 - (1 - 1/SRD)^K) \tag{23}$$

$$\text{where } K = ACI * LSTAVE\ .$$

3. Record condition ($ICR \geq 2$). A record condition is a conjunction of item conditions, i.e. two or more key names are involved. The search strategy in this case is to search the deepest level in the tree. Whenever a condition is satisfied, the item conditions for levels above are checked by following the *P-PTR*.

$$X_N = \frac{ACI}{2 * NKEY} SUMM(NODE(I - 1) * S(I)) + \frac{ACI * ICR}{2 * NKEY} SUMM(NODE(I - 1)) \tag{24}$$

$$X_D = SRD(1 - (1 - 1/SRD)^K \tag{25}$$

$$\text{where } K = ACI * LSTAVE\ .$$

4. Query condition ($RCQ \geq 2$). A query condition is a disjunction of record conditions. This calls for a record condition search for each of the RCQ record conditions. The search is performed in a block by block manner, so that each index block needs to be accessed only once.

$$X_N = \frac{ACI * RCQ}{2 * NKEY} SUMM(NODE(I - 1) * S(I)) + \frac{ACI * ICR * RCQ}{2 * NKEY} SUMM(NODE(I - 1)) \tag{26}$$

$$X_D = SRD * (1 - (1 - 1/SRD)^K) \tag{27}$$

$$\text{where } K = ACI * RCQ * LSTAVE\ .$$

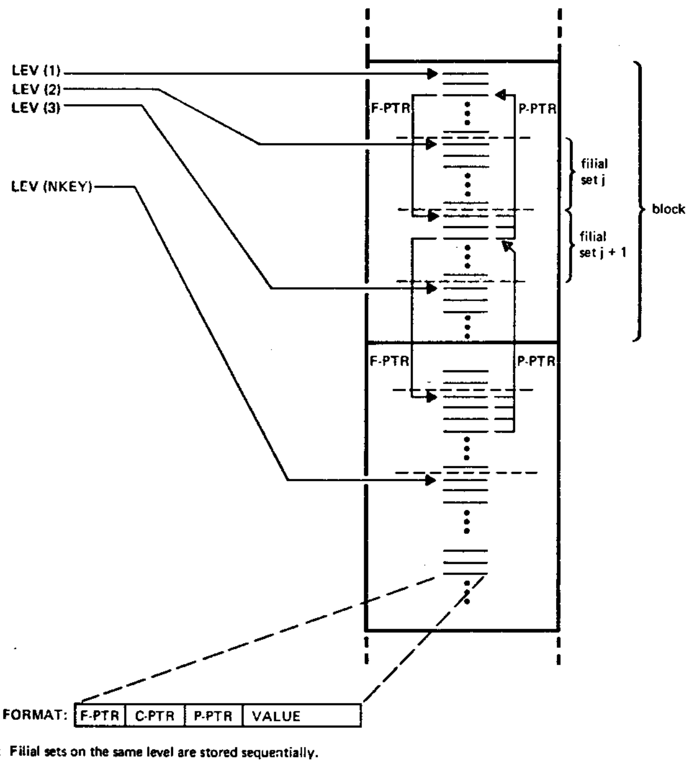


Fig. 6 Doubly-chained tree organisation, version 3

In summary, the following advantages and disadvantages can be attributed to versions 1 and 2 of a doubly-chained data base structure with the physical implementation traits illustrated in Fig. 4:

Advantages

1. Provide significantly better access time performance when the first K keys in the tree are specified in the queries or the number of item conditions per record condition is relatively large.
2. Easy to update. This stems from not allowing pointers to cross block boundaries and storing key values of a record in the same index block.
3. Low deterioration rate in the index blocks with updating.
4. Relatively easy to program.

Disadvantage

Performance is relatively poor in the case where arbitrary key names not corresponding to the first levels of the tree are used in the queries.

4. Doubly-chained structure version 3

In versions 1 and 2, the doubly-chained tree has been structured in such a way that key values taken from the same record are placed close together, as illustrated in Figs. 4 and 5. An alternate physical structuring is to place key values in the same filial set and on the same level close together. This strategy shall be called Version 3. Its layout is shown in **Fig. 6.** An entry point for each level of the tree is now used (each level corresponds to a key name). Starting from some entry point for some level, the nodes for the first filial set are stored sequentially in sort order. Following this are the nodes for the nextfilial set, and so on. The entry points are shown as $LEV(K)$. They could be stored in a file directory, or in the first block of the file. Their space requirement is negligible and thus it is disregarded in the derivations. Filial sets are denoted by the dashed lines.

As shown, this version requires the use of all three pointers. Whereas in versions 1 and 2 pointers were not allowed to cross boundaries, in this version they do. An example of an *F-PTR* and *P-PTR* that do and do not cross block boundaries is shown in Fig. 6. The *F-PTR*'s for the last level point to the special nodes through which records are reached. As in versions 1 and 2 the *VALUE* of a special node is a pointer to an overflow area (block); its *C-PTR* points to the first and its *P-PTR* points to the last of a string of consecutive data records. The special node is placed physically next to the corresponding last level node of the tree. The example in Fig. 2, 3 and **Fig. 7** illustrates and clarifies the structure.

The *C-PTR* of the nodes is heavily used for updates. At file generation time, the *C-PTR* points to the next sequential entry. If a node is inserted, it is placed in the reserve area and the *C-PTR*'s are modified so that it is logically in its correct place in the filial set.

As it turns out, most of the expressions to estimate storage and access time are the same as for versions 1 and 2, with the exception of the expressions for *BFI* and X_I. The expressions for version 3 are enumerated below and in the following paragraphs.

$$KLFIX = \left[\frac{KLAVE}{WORD}\right]_+ \tag{1}$$

$$BFI = \left[\frac{BLOCKW}{KLFIX + 3}\right]_- \text{ nodes/block} \tag{28}$$

$$RSI = \left[\frac{BFI}{10}\right]_+ \text{ reserve node space/block} \tag{3}$$

$$SRI = \left[\frac{NODE(NKEY) + SUMM\,(NODE(I))}{BFI - RSI}\right] \text{ blocks} \tag{4}$$

$$BFD = \left[\frac{BLOCKC}{RLAVE - (NKEY * KLAVE)}\right]_- \text{ records/block} \tag{5}$$

$$RSD = \left[\frac{BFD}{10}\right]_+ \text{ reserve record space/block} \tag{6}$$

$$SRD = \left[\frac{NREC}{BFD - RSD}\right]_+ \text{ blocks} \tag{7}$$

$$TOTSR = SRI + SRD \text{ blocks} \tag{8}$$

This version 3 is specifically designed for the situation in which the data base user requires the capability to search the data base on any specified level of the tree. This is the situation assumed in version 1.

The critical units of time are:

1. The time to process a node, T_N.
2. The time to access a block, T_T.

The average access time is again a function of the number of nodes processed X_N, the number of index blocks accessed X_I, and the number of data blocks accessed X_D (as in versions 1 and 2):

$$ACCTM = (T_N * X_N) + T_T(X_I + X_D)\ . \tag{9}$$

Determination of the values for X_N, X_I, and X_D is based on the level of logical complexity of queries.

1. Atomic condition ($ACI = 1$). It should be clear that the number of nodes and the number of data blocks that must be

LEV(I)	Pointer	Relative Address	Node F-PTR	Node C-PTR	Node P-PTR	Node Value
⋮	⋮	⋮	⋮	⋮	⋮	⋮
NKEY-2	105	105	106	–	–	M
NKEY-1	106	106	108	107	105	A ← Filial Set of M
NKEY	108	107	114	–	105	E ←
		108	109	110	106	B ←
		109	–	R1001	R1001	Overflow Pointer
		110	111	112	106	C ← Filial Set of A
		111	–	R1002	R1004	Overflow Pointer
		112	113	114	106	D ←
		113	–	R1005	R1005	Overflow Pointer
		114	115	–	107	F ← Filial Set of E
		115	–	R1006	R1006	Overflow Pointer
		116				
		⋮	⋮	⋮	⋮	⋮

Note: Each record or tuple (without the indexed keys) is stored in relative address Rxxxx.

Fig. 7 Physical structure of the doubly-chained organization, version 3 Fig. 6, for the data in Fig. 2.

Table 3 Characteristics of the test data bases

	Data base 1	2	3	4	5	6
Group 1						
NREC	3,676	3,676	5,239	18,573	15,888	1,296
NKEY	6	4	4	4	6	4
DKV	2,914	496	483	1,175	271	466
TNODES	7,265	2,082	2,251	8,454	2,090	673
LSTAVE	8	30	43	63	352	11
Group 2						
RLMIN-RLMAX	54-166	54-166	52-166	50-329	217-479	141-1,186
RLAVE	87	87	93	84	236	404
Group 3						
KLMIN-KLMAX	0-77	2-7	2-7	2-7	1-10	1-17
KLAVE	10	3	4	3	4	4

NREC = number of records
NKEY = number of keys doubly-chained
DKV = number of distinct key values
TNODES = total number of nodes
LSTAVE = average list length
RLMIN = minimum record length
RLMAX = maximum record length
RLAVE = average record length
KLMIN = minimum key-value length
KLMAX = maximum key-value length
KLAVE = average key-value length

accessed is the same as in version 1 and that discussion will not be repeated here. The number of index blocks, however, may differ significantly. The search strategy for an atomic condition involves, first, obtaining the address of the block containing the first filial set for the specified key name. Since this block, from some point on, contains only nodes corresponding to the specified key name, fewer index blocks must be accessed. As usual, the average is used.

$$X_I = \left[\frac{SRI}{NKEY}\right]_+ \tag{29}$$

$$X_N = \frac{1}{2 * NKEY}\, SUMM(NODE(I-1) * S(I)) \tag{20}$$

$$X_D = SRD * (1 - (1 - 1/SRD)^K) \tag{21}$$

where $K = LSTAVE$.

2. Item condition ($ACI \geq 2$). Since nodes corresponding to a single key name are still being accessed, X_1 remains the same:

$$X_I = \left[\frac{SRI}{NKEY}\right]_+ \tag{30}$$

Table 4 Detailed characteristics (statistics) of the test data bases

	Data base 1			*Data base 2*			*Data base 3*			*Data base 4*			*Data base 5*			*Data base 6*		
I	N_I	S_I	F_I	N_I	S_I	F_I	N_I	S_I	F_I	N_I	S_I	F_I	N_I	S_I	F_I	S_I	S_I	F_I
0	1	—	—	1	—	—	1	—	—	1	—	—	1	—	—	1	—	—
1	2	2	1	2	2	1	2	2	1	2	2	1	9	9	1	16	16	1
2	32	16	2	32	16	2	32	16	2	32	16	2	40	4·44	9	34	2·13	16
3	163	5·09	32	159	4·97	32	131	4·09	32	427	13·34	32	115	2·87	40	156	4·59	34
4	524	3·21	163	1,889	11·88	159	2,086	15·92	131	7,993	18·72	427	209	1·81	115	467	2·99	156
5	2,881	5·50	524										379	1·81	209			
6	3,663	1·27	2,881										1,338	3·53	379			
N	1,210·8			520·5			562·7			2,113·5			348·3			168·2		
S	5·51			8·71			9·50			12·51			3·91			6·42		
F	600·5			48·5			41·5			115·5			125·5			51·8		
TNODES	7,265			2,082			2,251			8,454			2,090			673		

K = number of indexed access key-names $\quad K = NKEY$

N_I = number of nodes on level $I (N_0 = 1)$ $\quad N_I = NODE(I)$

S_I = average size of filial sets on level I $\quad S_I = \frac{N_I}{N_{I-1}}$

F_I = number of filial sets on level I $\quad F_I = N_{I-1}$

N = average number of nodes on a level $\quad N = \frac{1}{K} \sum_{I=1}^{K} N_I$

S = average size of filial sets $\quad S = \frac{1}{K} \sum_{I=1}^{K} \frac{N_I}{N_{I-1}}$

F = average number of filial sets on a level $\quad F = \frac{1}{K} \sum_{I=1}^{K} N_{I-1}$

TNODES = total number of nodes $\quad TNODES = \sum_{I=1}^{K} N_I$

$$X_N = \frac{ACI}{2 * NKEY} SUMM(NODE(I-1) * S(I)) \quad (22)$$

$$X_D = SRD(1 - (1 - 1/SRD)^K) \quad (23)$$
where $K = ACI * LSTAVE$.

3. Record condition ($ICR \geq 2$). In this case, a logical conjunction is implied which contains more than one key name. The average number of key names is given by *ICR* (see Appendix 1). The assumed search strategy is to search on the lowest level of the tree. Each time a condition is satisfied, the *P-PTR* and *F-PTR* are saved. When the level is exhausted, one will have obtained a set of *P-PTRS* for resolving conjuncted conditions and a set of *F-PTRS* for obtaining data-record addresses. Because of the level by level clustering technique, there is a high probability that most of the *P-PTRS* and *F-PTRS* reference the same block. Also, many of the *P-PTRS* can be discarded on the level above. For this reason it is reasonable to assume that for each item condition satisfied, no more than *ICR* additional blocks need be accessed.

$$X_I = ICR \left[\frac{SRI}{NKEY}\right]_+ \quad (31)$$

$$X_N = \frac{ACI}{2 * NKEY} SUMM(NODE(I-1) * S(I))) + \frac{ACI * ICR}{2 * NKEY} SUMM(NODE(I-1)) \quad (24)$$

$$X_D = SRD(1 - (1 - 1/SRD)^K) \quad (25)$$
where $K = ACI * LSTAVE$.

4. Query condition ($RCQ \geq 2$). In this case, several (RCQ) record condition searches are performed simultaneously keeping track of the pointers which correspond to each record condition.

$$X_I = ICR \left[\frac{SRI}{NKEY}\right]_+ \quad (32)$$

$$X_N = \frac{ACI * RCQ}{2 * NKEY} SUMM(NODE(I-1) * S(I)) + \frac{ACI * ICR * RCQ}{2 * NKEY} SUMM(NODE(I-1)) \quad (26)$$

$$X_D = SRD * (1 - (1 - 1/SRD)^K) \quad (27)$$
where $K = ACI * RCQ * LSTAVE$.

The following advantage and disadvantages may be attributed to version 3, whose physical implementation characteristics are illustrated in Figs. 6 and 7:

Advantage
Provides good search performance when arbitrary key names appear in the queries and the number of them is small.

Disadvantages
1. File generation, search and update routines are relatively intricate and difficult to program.

2. High deterioration rate in index blocks with updating.

5. Test results and evaluation

The performance of the alternative doubly-chained organisations that have been modelled in the two previous sections will be analysed with the aid of six sample data bases. The data bases used were real life files containing information on Naval missile systems and test equipment. The size of these data bases ranged from 1296 records to 18,573 records. Although they are rather small to medium in size, they provide a practical and valid test basis for purposes of comparison and discussion. The characteristics of the data bases are summarised in **Table 3.** The more detailed statistics are enumerated in **Table 4.** These statistics are based on measurements of the data bases taken by means of a special program written in ANSI COBOL (Cardenas, 1973).

The three alternative doubly-chained strategies are: (1) Version 1, structured as illustrated in Figs. 4 and 5, in which the K key names appearing in a query do not correspond to the first K levels of the directory, that is, queries contain arbitrary combinations of index keys to access the data base—as long as K is a subset of the *NKEY* directory keys (the indexed key names); (2) Version 2, structured as illustrated in Figs. 4 and 5, in which the doubly-chained directory structure and the K key names appearing in a query are such that the key names correspond to the first K levels in the directory structure; (3) Version 3, structured as illustrated in Figs. 6 and 7, and in which, as in version 1, queries contain arbitrary combinations of index keys to access the data base—as long as K is a subset of the *NKEY* directory keys.

Storage requirements

Table 5 summarises the total storage requirement *TOTSR* for each of the data bases under the three versions of the doubly-chained organisation. The equations derived in the two previous sections are the formulations used for estimating the storage requirement *TOTSR*. *TOTSR* is the same for versions 1 and 2, since the difference between these versions is in the strategy to search the directory. *TOTSR* for version 3 is only slightly more than for these versions. The difference is rather small, as can be seen by comparing the corresponding equations for *BFI* and *SRI* making up *TOTSR* in Sections 3 and 4. These comparisons assume a similar order of the *NKEY* keys from level 1 to level *NKEY* in the directory.

It is important to note that the total storage required in doubly-chained data organisations may be in fact less than that taken up by the original hierarchical record structure, or the tabular or relational data structure. This can be visualised by examining in Fig. 2 the original values for a subset of (*a*) key names of a set of records, or (*b*) domains of tuples of a relation, and realising that in Fig. 3 they are represented by only seven nodes, compared to the original 18 entries. Although the storage for the pointers is an added expense, the net saving increases with the number of key names or domains (recall that each key name, or domain of a relation, corresponds to a level in the tree) and with the repetition of values for the same key name or domain. This is dramatised by realising that if the subset of data in Fig. 2 were the same from row to row, the doubly-chained tree would contain only three nodes. In the extreme case that no repetition of values occurred between rows, the tree would contain at worst as many nodes as there are values, plus pointer space for each node; no storage saving would be realised under these extreme and unlikely conditions.

Table 5 Total storage requirements* for the doubly-chained test data bases

Doubly-chained structure	*Data base* 1	2	3	4	5	6
Version 1	39	37	51	169	359	57
Version 2	39	37	51	169	359	57
Version 3	44	39	53	179	361	57

*Expressed in UNIVAC FASTRAND blocks or tracks—10,740 characters per block.

The storage saved by the doubly-chained organisation, S_s, is approximately:

$$S_s = NREC * NKEY - 3 * TNODES \quad (33)$$

taking a pessimistic view that the storage required by each of the three pointers is the same as for the key value of a node. Note that the number of nodes, *TNODES*, may be greater than the sum of the number of distinct key values of the *NKEY* key names, that is,

$$TNODES \geq \sum_{I=1}^{NKEY} NVAL(I) \ . \quad (34)$$

The reason is that while within a filial set key values are unique, between filial sets on the same level of the tree values may be repeated.

The Group 1 measurements on Table 3 show savings in storage ranging from only 261 key spaces for data base 1 to 89,058 key spaces for data base 5. The saving depends very much on the characteristics of the contents of the data base and on the selection and ordering of the key names in the directory. The main point here is to expose the potential savings that may be realised by doubly-chained trees in comparison with, for example, inverted data organisations in which any inversion entails additional storage (since records must preserve the indexed key values, unlike doubly-chained structures).

Access time to answer a query

Estimation of the average time *ACCTM* to answer a query is more complex. The complexity of queries is an added parameter that affects it. *ACCTM* is obtained from Equationf(9):

$$ACCTM = (T_N * X_N) + T_T(X_I + X_D) \text{ seconds} \quad (9)$$

T_T is the average time to access a block and T_N is the average time to process a node. The detailed expressions for X_N, the number of nodes processed; X_I, the number of index blocks accessed; and X_D, the number of data blocks accessed, have been derived in Sections 3 and 4.

For a given data base and specific query, the average number of data blocks accessed X_D should be really the same no matter what data base organisation is used. However, in the models derived in Sections 3 and 4 this is not necessarily the case. The reason is that for version 2 X_D is estimated via equations (15) and (18), while for versions 1 and 3 X_D is estimated via equations (21), (23), (25) and (27). The difference in these two sets of estimators is that the former equations utilise the knowledge that the number of records that may potentially satisfy the query is $RCQ * \dfrac{NREC}{NODE(ICR)}$. This consideration does not hold for version 1 and 3, and thus the resort is to Equation (11) which estimates coarsely the average list length *LSTAVE* over all *NKEY* keys (whether or not all keys in fact appear in the query). An obvious refinement would be to estimate the average list length for each access key.

Table 6 summarises the average access time for the six data bases. The device parameters used are $T_T = 100$ msec and $T_N = 1.5$ msec. The estimates are shown for each of the doubly-chained organisation strategies, for four levels of query complexity. The timings illustrate a very significant performance advantage of version 2. Version 3 shows only slightly better access time than version 1, for low levels of query complexity.

Table 6 Average access time† by doubly-chained tree organisation strategy and query complexity

*Query complexity**	*Doubly chained organisation***	*Data base* 1	2	3	4	5	6
0	Version 1	4.3	3.1	4.1	9.8	23.4	1.4
0	Version 2	2.0	2.3	3.3	6.7	22.7	1.2
0	Version 3	2.1	2.5	3.5	7.5	22.9	1.2
1	Version 1	8.4	5.7	7.2	23.4	37.7	4.4
1	Version 2	2.0	3.3	4.7	14.0	35.4	3.8
1	Version 3	6.0	5.1	6.6	21.1	37.1	4.2
2	Version 1	17.4	6.4	7.8	25.1	39.5	5.2
2	Version 2	2.2	0.7	0.9	2.1	7.9	0.6
2	Version 3	16.5	6.4	7.8	25.1	39.3	5.2
3	Version 1	112.2	25.1	27.0	94.2	66.5	17.0
3	Version 2	3.3	1.3	4.0	5.5	36.1	3.4
3	Version 3	111.0	25.1	26.5	94.2	66.3	17.0

†All times are expressed in seconds
*As per Appendix 1:
Query complexity = 0—Atomic condition only ($ACI = 1$)
Query complexity = 1—Item condition only ($ACI = 5$)
Query complexity = 2—Record condition only ($ICR = 4$)
Query complexity = 3—Query condition only ($RCQ = 8$)
**Primary difference between (*a*) Versions 1 and 2 and (*b*) Version 3 is in physical structure.
**Primary difference between (*a*) Versions 1 and 3 and (*b*) Version 2 is in whether or not the set of K access key names in a query corresponds to the first K tree levels.

Recall that the main difference between (*a*) versions 1 and 2, and (*b*) Version 3 is in physical organisation, or storage structure, as illustrated by contrasting Figs. 4 and 5 with Figs. 6 and 7. The main difference between (*a*) Versions 1 and 3, and (*b*) Version 2 is in the ordering and strategy for searching nodes.

The results in Table 6 illustrate that the best possible performance for the doubly-chained tree is indeed quite good if the K key names appearing in the query in any arbitrary combination correspond to the first K doubly-chained levels. This implies a priori knowledge of the types of queries to order the *NKEY* levels optimally. If optimality due to ordered search of the first K levels cannot be achieved, perhaps because of the variability of user's queries, then only little improvement in access time can be achieved through such physical structuring strategies. However, it is speculated that for higher degrees of double-chaining, that is, for larger *NKEY*, the relative increase in access time for version 3 is possibly less than for versions 1 and 2.

A close examination of the relative contribution of the node processing (CPU) component $T_N * X_N$ and of the data transfer (I/O) component $T_T(X_T + X_D)$ of the access time *ACCTM* Equation (9) for each of the sample data bases for various query complexities shows that:

1. The relative contribution of node processing time for versions 1 and 3 is small for the simpler queries and it becomes as significant as data transfer for queries involving two or more key names (query complexity = 2). For very complex queries of type 3, node processing is by far the largest contributor of access time.
2. The relative contribution of data transfer is the largest portion of access time in version 2. However, node processing time is not negligible and it cannot be discarded, particularly at the very high query complexity level 3 when it becomes more significant (but not the largest contributor).

The details of X_I, X_D and X_N for the six data bases for the four types of queries would require several tables and hence are not included here.

The analysis has thus shown that the determination of whether a given doubly-chained data base system is CPU or I/O bound depends on both the search strategy and the specific application. It is not correct to say flatly that the bound is I/O, although the tendency of data base folklore (as opposed to fact) is to consider I/O as the bottleneck.

Minimisation of the search space or the number of nodes to be examined to locate the pertinent nodes should be a design goal for doubly-chained data bases. If the nature of queries is so variable that version 2 strategy, which minimises the search space, cannot be used, then it is suggested that performance-enhancing strategies of the following types, which do not depend on query specifications, be used, namely:

1. Strategies that decrease the number of nodes to represent the tree structured file. There is a certain order or permutation for the key names in the directory such that the number of nodes (key values) is minimised. It is directly dependent on the relative distribution of key values, i.e. on the specific data base. Significant savings can be obtained. This is illustrated in Appendix 2. Rotwitt and deMaine describe the approach to optimising storage (1971).
2. Purely physical organisation strategies, of the type exemplified by versions 1 and 2 versus version 3, in Figs. 4, 5, 6 and 7. In effect, they attempt to reduce the amount of index block transfers from secondary to primary storage, and hence the search space in main core for the pertinent directory nodes.

As illustrated through the six test cases, enhancements of the second type appear to result only in rather small improvements in access time and storage requirements. Type 2 physical structure alternatives entail different programs in any high level language (COBOL, PL/I, etc.) for generation, retrieval and update of the data base. Thus, it is unlikely that a shift from version 1 to version 3, or vice versa, can be justified. Changes in software are very costly. On the other hand, a large portion of the software for generation, retrieval and update would be common in versions 1 and 2. Furthermore, the same basic routines to structure or restructure the order of the directory in an attempt to achieve the gains of version 2 strategy could also be used to achieve gains of type 1 above if version 2 strategy is not possible.

At the end of Sections 3 and 4 it was concluded that (*a*) file generation, search and update routines are relatively intricate, and (*b*) there is a high deterioration rate in index blocks with updating in version 3 compared with versions 1 and 2. This is in spite of the fact that conceptually version 3 exemplified in Figs. 6 and 7 seems to be more easily visualised initially. In summary, it is concluded that version 3 is not a practical choice. The strive should be to achieve the significant performance gains of version 2.

6. Conclusions

The doubly-chained data base organisation has been analysed and modelled to derive expressions for average access time and storage requirement, taking into account the influence of the contents of the data base, query complexity, device and processor time specifications, and implementation-oriented characteristics. These factors have a large influence on access time performance. Expressions for three alternative doubly-chained strategies, versions 1, 2 and 3, have been derived. They represent alternative search strategies and physical structuring. The strategies have been compared and evaluated with the help of six real life data bases.

Storage requirements do not vary significantly among doubly-chained strategies. However, storage is significantly less compared to other data base organisations such as inverted files. Double-chaining will usually result in less storage than the original sequential or tabular structure; in contrast, any inversion involves additional storage.

Average access time varies significantly. The average access time for version 2 is significantly less than for versions 1 and 3, by several orders of magnitude depending of course on the specific data base and query—as illustrated by the six test data bases. An important first order factor affecting performance is the order of the access keys in the doubly-chained tree with respect to the set of access keys K in the query; it is in this regard that version 2 differs from versions 1 and 3. If the K key names appearing in a query correspond to the first K levels of the doubly-chained tree, the highlight of version 2, then access time is minimised.

Variations in the physical structuring of the doubly-chained organisation, as exemplified by (*a*) Versions 1 and 2 versus (*b*) Version 3, affect storage and access time very little. However, the physical structuring of version 3 is such that its generation, search and updating is relatively intricate, and updating tends to cause high deterioration in the index blocks. Hence, it is not an attractive choice if it is also realised that the similarity of versions 1 and 2 is such that much of the generation, search and update software would be in common. Thus a shift from version 1 to 2 would be practical, but not from version 3.

If the nature of queries is so variable, logically speaking, that version 2 strategy cannot be used, then it is suggested to use strategies that reduce the number of nodes by ordering them appropriately, independent of query considerations. A reduction in the number of nodes is effectively equivalent, from the point of view of access time, to limiting node search to a subset of the total node space (the essence of version 2).

If both queries and data base contents are very dynamic, then the advantages of neither version 2 nor reduction of node space can be achieved practically. As the number and frequency of occurrence of key names in the queries increases, the degree of indexing should be increased. Thus, if all key names have equal probability of appearing in queries, then complete double-chaining of the data base may be warranted. The determination of the degree of indexing is an important issue that has not been addressed directly. This is an area of future work. The analysis presented is essential toward this end.

An important insight in the analysis presented is that it cannot be stated without careful qualification that a doubly-chained data base system is either I/O (data transfer) bound or CPU (node processing) bound, although the tendency of data base folklore (as opposed to fact) is to view I/O as the bottleneck. The search strategy (versions 1 and 3 versus version 2), the specific contents of the data base and the complexity of queries determine whether the bound is I/O or CPU. The six test data bases do show that for low query complexities data transfer is the main contributor of total access time. As query complexity increases, the relative node processing time increases and may be the largest contributor.

The previous sections provide the matter leading to the conclusions in the previous paragraphs. Although it is acknowledged that implementation-specific aspects may have underestimated effects on performance, the analysis presented takes into account real life logical-physical aspects as well as physical structuring aspects. Thus, it is not relegated to the logical and conceptual realm. Needed insights and quantitatively based approaches for doubly-chained data base design have been presented. Quantitative approaches and well proven guidelines are much needed to improve the still rudimentary and ad hoc practice of data base system architecturing and implementation.

Acknowledgement

Many thanks are due to Drs. K. Eswaran, S. P. Ghosh, W. F. King and C. P. Wang of the IBM Research Laboratory, San Jose, California and M. E. Senko, IBM T. J. Watson Research Laboratory, Yorktown Heights, New York, for their valuable discussions and suggestions toward clarifying various issues.

Appendix 1

The scheme used for classifying queries according to complexity is the following.

1. An *atomic condition*, A, will have the form

$$NAME \left\{ \begin{matrix} < \\ = \\ \neq \\ > \end{matrix} \right\} VALUE$$

where $NAME$ is the item name or key name in the COBOL-record sense, or the domain in the tabular or relational sense.

2. An *item condition*, I, is a disjunction of atomic conditions, A_1 OR A_2 OR . . . OR A_L, such that each A_i reflects the same item name (key name or domain). ACI is defined as the number of atomic conditions per item condition I.
Example: AGE = 20 OR AGE = 21 where ACI = 2

3. A *record condition*, R, is a conjunction of item conditions I_1 AND I_2 AND . . . AND I_M such that each I_j reflects a distinct item-name (key name or domain). ICR is defined as the number of item conditions per record condition R.
Example: (AGE > 20) AND (SEX = FEMALE)

where $ICR = 2, (ACI_1 = ACI_2 = 1)$

4. A *query condition*, Q is a disjunction of record conditions, R_1 OR R_2 OR . . . OR R_N. RCQ is defined as the number of record conditions per query condition Q.
Example: [(AGE > 18) AND (SEX = FEMALE)] OR [(AGE > 20) AND (SEX = MALE) AND (CLASS = SENIOR OR CLASS = GRADUATE)]

where $RCQ = 2;$
$ICR_1 = 2, (ACI_1 = ACI_2 = 1);$
$ICR_2 = 3, (ACI_1 = ACI_2 = 1, ACI_3 = 2);$

These definitions are inclusive in the sense that an atomic condition is also an item condition, a record condition, and a query condition. Practically all types of queries can be placed into the above formats.

Appendix 2

Strategies to decrease the number of nodes to represent tree structured files have been proposed in the past (Rotwitt and deMaine, 1971). There exists a permutation of the key names of records (or domains of a relation) for which the number of nodes in the tree is minimal. As an example, consider the sample

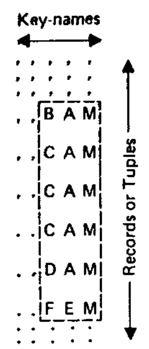


Fig. 8 Sample data values

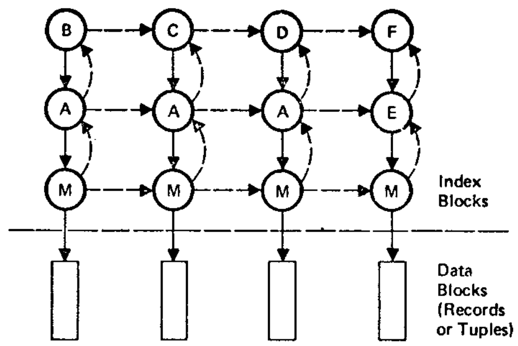


Fig. 9 Doubly-chained tree organisation for the data in Fig. 8.

values in Fig. 2 and its doubly-chained representation in Fig. 3. A different ordering of key names as in **Fig. 8** results in the 12 nodes in **Fig. 9**, compared with only 7 nodes in Fig. 3. Thus, through techniques based on the relative distribution of key values, it is possible to minimise the number of nodes, thus minimising the average access time for queries with arbitrary combinations of key names (assumed to be indexed). This optimisation of nodes is highly dependent on the distribution of key values. Thus, close examination of key distributions and software means to reorder key names (domains) in the tree representation are necessary in very dynamic data base environments to obtain the potential benefits of such node reduction strategies.

References

CARDENAS, A. F. (1973). Evaluation and Selection of File Organisation—A Model and System, *CACM*, Vol. 16, No. 9, pp. 540-548.
CASEY, R. G. (1973). Design of Tree Structures for Efficient Querying, *CACM*, Vol. 16, No. 9, pp. 549-556.
CODD, E. F. (1970). A Relational Model of Data for Large Shared Data Bases, *CACM*, Vol. 13, No. 6, pp. 377-387.
DODD, G. D. (1969). Elements of Data Management Systems, *Computing Surveys*, Vol. 1, No. 2, pp. 117-133.
HU, T. C. (1972). A Comment on the Double-Chained Tree, *CACM*, Vol. 15, No. 4, p. 276.
IBM CORP. (1975). *IBM System/360 Operating System, PL/I Programmer's Guide*, Form C28-6594.
KING, W. F. (1974). On the Selection of Indices for a File, IBM Research Laboratory Report RJ1341, San Jose, California.
KNUTH, D. E. (1969). *The Art of Computer Programming*, Vol. 1, Addison-Wesley Publishing Co.
LEFKOVITZ, D. (1969). *File Structures for On-Line Systems*, New York, Spartan Press.
PATT, Y. (1969). Variable Length Tree Structures Having Minimum Average Search Time, *CACM*, Vol. 12, No. 2, pp. 72-76.
ROTWITT, T., and DEMAINE, P. A. D. (1971). Storage Optimization of Tree Structured Files Representing Descriptor Sets, *Proceedings, ACM SIGFIDET Workshop*, San Diego, California, November 11-12, 1971.
SALTON, G. (1968). *Automatic Information Organization and Retrieval*, McGraw-Hill Book Co., New York.
STANFEL, L. E. (1970). Tree Structures for Optimal Searching, *JACM*, Vol. 17, No. 3, pp. 508-517.
SUSSENGUTH, E. H. (1963). The Use of Tree Structures for Processing Files, *CACM*, Vol. 6, No. 5, pp. 272-279.
WEINBERG, G. M. (1970). *PL/I Programming: A Manual of Style*, McGraw-Hill, New York.

Book review

Modelling in data base management systems, edited by G. M. Nijssen, 1976; 418 pages. (*North-Holland*, US $35.00)

This book contains the proceedings of the conference organised by the IFIP Technical Committee 2, at Freudenstadt, Germany in January 1976. There are now at least three groups (IFIP TC2, ACM SIGMOD/SIGFDT, VLDB) presenting annual conferences on data base management with follow up publications of proceedings. Organisations undertaking data base research will want copies of all these publications in their libraries. It is unfortunate, therefore, that IFIP, while making use of camera-ready copy, choose to publish in a glossy hardback form at a price of $35.00.

The conference had several inter-related themes, dominated by consideration of data base design at the conceptual level. Various approaches to the modelling process and to the specification of formal conceptual schema languages are presented. Brachi *et al* offer the 'binary logical association' as a modelling tool. Several papers, e.g. Hall *et al*, make a case for the relational approach to conceptual data base design. Kalinchenko discusses the mapping between the relational model and the CODASYL network model to reinforce the widely held view that the former is appropriate at the conceptual schema level and the latter at the external schema level.

Many of the papers mention DBMS architecture either as the context for the conceptual schema or as a subject in its own right. Nijssen gives his view of a gross architecture for the next generation of DBMS. Most authors relate their ideas to the architecture originating from the ANSI SPARC committee. Senko goes as far as claiming that DIAM is an example of this architecture.

The other major theme is the integrity facilities provided by various approaches to data base management. It is surprising to find that several papers cover similar ground to previously published work, without reference. Weber criticises the relational model, and Engles the CODASYL specification, for not providing adequate consistency controls for concurrent data base access. Gray *et al* deal with locking levels and assess IMS/VS (a hierarchical system) and DMS/1100 (a CODASYL network system).

The papers presented at the conference provide an up to date view of many of the debates within and among the different approaches to data base management.

J. S. KNOWLES (Aberdeen)