DIRECTED GRAPH STRUCTURES FOR DATA BASE MANAGEMENT; THEORY, STORAGE STRUCTURES AND ALGORITHMS

by

Frank Barnett Ray

November 1972 TSN-31

This paper constituted the author's dissertation for the Ph.D. degree at The University of Texas at Austin, December 1972.

THE UNIVERSITY OF TEXAS AT AUSTIN

COMPUTATION CENTER

TABLE OF CONTENTS

LIST OF FIGURES

INDEX OF SPECIAL TERMS

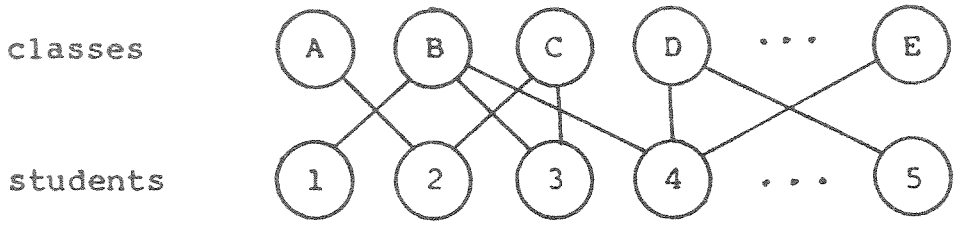
TERM PAGE

TERM PAGE

CHAPTER I

INTRODUCTION

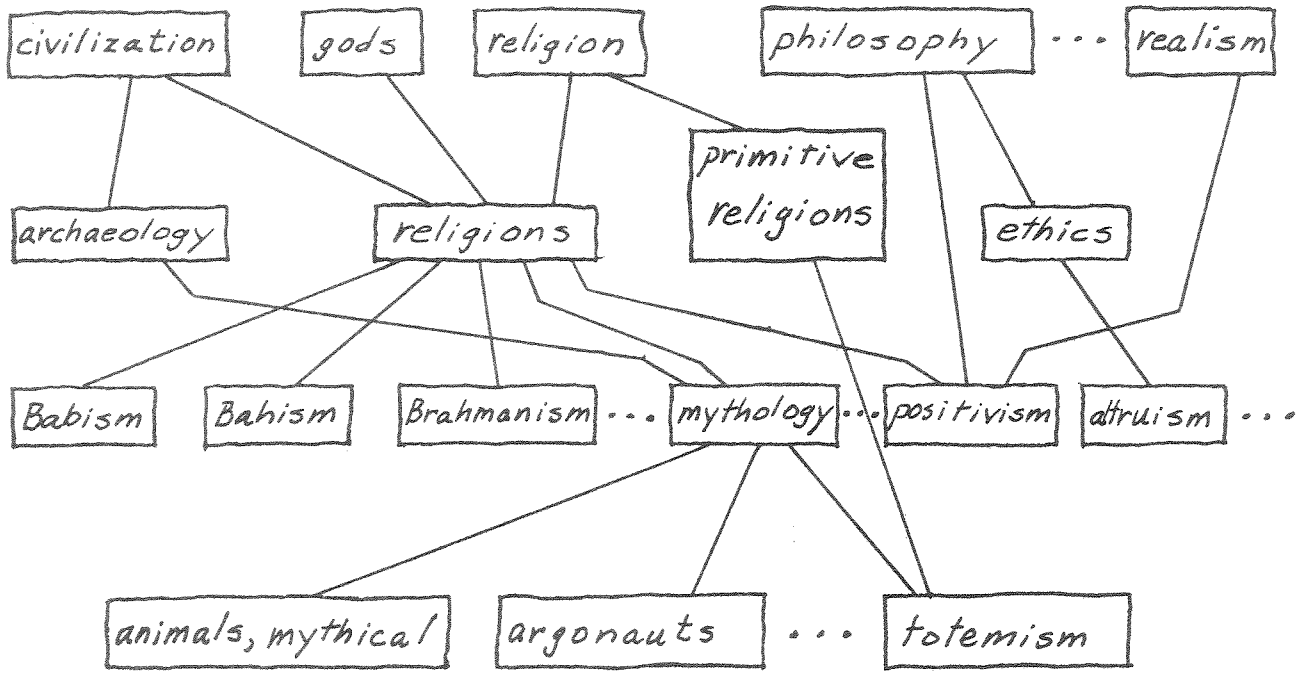
The massive April 1971 Report of the CODASYL Data Base Task Group (C4), in setting guidelines for future data management systems, includes a section on network-related data bases, in recognition that "natural" data relationships exist which may not be economically represented by structures with less richness than that possessed by a graph. The CODASYL Report does not dwell on implementation nor on the explicit types of graphical representation. It does, however, include an archetypal example of school children and classes, pointing out the fact that one class containing many students produces a one-to-many relationship of each class to students, and that one student may be in many classes produces a one-to-many relationship of each student to a number of classes. The suggestion of this "natural" example is to group classes as a set of labelled "nodes", group students likewise, and make an "edge" connection between classes and students to indicate the relationships. Without defining specifically what is meant by "node" and "edge", the following diagram would be a good representation of the structure:



This diagram indicates that class A consists at least of student 2, and student 2 is in classes A and C, class B consists at least of students 1, 3 and 4, etc.

This simple two-group example may obviously be separated into a "hierarchy" in which one group is placed above the other. It is also possible to introduce a third category for the above example which naturally would be separated by "classes" from "students". Consider the category of subject areas, i.e., science, art, social science, etc. Several "classes" fall into one subject area and it is reasonable to relate a student's involvement in a subject area via his class involvement; hence the natural separation of students and subject areas by classes.

While there are natural data structures which do not easily separate into hierarchical groupings, and structures which allow cyclic relationships to occur, it is with the various natural data structures which do separate into hierarchical groupings that this paper is concerned. The following example regarding the classification of documents by subject has been derived directly from information selected from the Library of Congress Subject Headings list (Q1), and is logically hierarchical:



Throughout this list, there is a rough idea of hierarchy, for example, archaeology is more specific than civilization, altruism "comes under" ethics, totemism is a part of primitive religion and primitive religion is "under" religion.

In more commercial areas, data base designers are faced with problems such as the purchase-order/vendor problem given as an example for the Integrated Data Store System (B1), where for each item purchased there are several vendors, and for each vendor, there are several items, which produces a common natural data structure for the demonstration of the Integrated Data Store. This example may be readily expanded to one which appears later in this paper to a manufacturing situation involving suppliers, parts, subassemblies, components and buyers. Here the hierarchy is effectively established by the three cate-

gories, parts, subassemblies, and components, because parts collectively form subassemblies, according to the manufacturer, and subassemblies collectively form components. For the other two categories, since suppliers supply parts and buyers buy components, these categories belong at the "parts" and "components" ends of the data hierarchy, respectively. Thus the whole hierarchy could be diagrammed as follows:

Suppliers

Parts

Subassemblies

Components

Buyers

A detailed example of a data structure constructed on this particular hierarchical basis is presented in Chapter V.

A great many natural data structures exhibit a hierarchy, and it would be pointless to ignore this if it exists, for even a primitive ordering of data can be useful when organizing it within a data management system. Special cases of hierarchical data have been shown to be readily imbedded in "tree" structures, and the use of "tree" data structures in data management applications is well known today (L1:Landamer, L4:Lowenthal, S6, D1:D'Imperio, H2:Hardgrave). Several attempts have been made to generalize both hierarchical and non-hierar-

chical data structures to allow richer and more varied associations of data entities than those which are allowed by tree structures. But although computer systems have been designed to manipulate graphical data structures, they are, in a practical sense, too general in scope for data management application.

The GRASPE system, for example, developed at the University of Texas at Austin (P1:Pratt and Friedman), has been constructed by first specifying the semantics of a basic "core" of graph operations, and then extending a high-level computer programming language (LISP 1.5) to define the syntax of GRASPE and the operations on its graphs. In theory, one could construct a data management system in GRASPE, but it is possible that the complete generality of such a system would make it so slow it would be commercially useless, although perhaps an interesting tool to study network problems with. The set of functions on graphs provided by Pratt and Friedman does, however, supply the experimenter with a handy superset of data management operations.

The General Electric-Honeywell Integrated Data Store System, or IDS, (B1:Bachman, I1), a semi-automatic[1]

---

[1]The adjective "semi-automatic" seems applicable to IDS because no attempt is made to implicitly order the data sets making up the chains in a way which would decrease the storage required, decrease the retrieval times, and appear totally transparent to the user. A truly automated data management system should require very little of its user

file management system, is somewhat less general than the GRASPE system. Here the semantics of particular kinds of operations on particular kinds of graphs are defined. A very significant feature in the system is the use or the chain, a form of directed graph, as a basis for the abstract data structures. As Bachman (B1:Bachman,C., Integrated Data Store) states,

> The task of organizing data records for meaningful association... is achieved through the use of chains, which provide cross-referencing linkage between records.

The integrated Data Store thus attempts to supply the user with a more powerful graphical data structure than a "tree" can provide, by using explicit linkage information in the form of pointers to construct chains.

In the search for other attempts to represent data structures more general than trees, two interesting papers on the organization of semantic data provide a valuable insight. Stanley Su (S5) uses an elaborate associative net to connect data items and to define semantic relations, as does Ross Quillian (M2:Minsky), although these appear in greatly differing forms. The following quote from Quillian's paper in Semantic Information Processing brings to light the significance of the data network, and although it was written in the context

---

in the way of his knowledge of internal orderings and yet still provide retrieval times he can tolerate.

of semantic information processing, today it has a strong bearing on the more sophisticated data management systems:

> As anyone who has ever reorganized a paper several times will realize, an outline organization is only adequate for one hierarchical grouping, when in fact the common elements existing between various meanings of a word call for a complex cross-classification. In other words, the common elements within and between various meanings of a word are many, and any one outline designed to get some of these together must at the same time separate other common elements, equally valid from some other point of view. Making the present memory network a general graph rather than a tree (the network equivalent of an outline) and setting up tokens as distinct nodes makes it possible to loop as many points as necessary back into any single node and hence in effect to show any and every common element within and between the meanings of a word.

Quillian's comments about cross-classification are equally valid for several noteworthy examples of natural data structures in which automated processing is already desirable and is rapidly becoming an economic necessity. Libraries, in general, and large libraries such as the U.S. Patent library and the Library of Congress in particular depend heavily on the structural richness of cross-classification. Any librarian would feel severely handicapped if the data structure in his subject card catalog were limited to a tree structure. Large-scale industrial and commercial inventories are frequently plagued with the problem of imposing a tree-like data structure on data which is in fact heavily cross-linked. The technique of mass production and the interchangeability

of mechanical parts has made it possible for individual parts and whole subassemblies of parts to be used (cross-classified, therefore) in different, larger assemblies. Indeed, the success or failure of an industrial designer may well depend upon his ability to recognize the existence of some usable entity already designed for something perhaps quite removed from his immediate sphere of activity, and this ingenuity must eventually be recorded in some inventory.

Orderly arrangements of knowledge, in libraries, industrial systems, commercial records, and public records, are imbedded by their designers within data structures in order to decrease updating and search times. These structures usually involve a hierarchy of some sort, cross-indexing, and many forms of data-oriented sorting. This is true whether the systems are implemented with machines or a host of clerks. Both hierarchical and data-oriented ordering are methods used to speed the various functions of the data management. Linkages, indexing, and cross-relating are enhancements to the data structure which pay off in terms of data retrieval speed, although they cost in terms of storage space.

The widely-used tree structure is adequate for hierarchical data groupings which have a characteristic tree form, for example, data organized by cities within a county, by counties within a state, by states within a

region, by regions within a country, etc. However, the tree structure is inadequate for the types of data structure indicated by the various examples noted above because adjacent hierarchies may have many kinds of interconnections.

The research presented here is the result of the search for formalisms which will lead to efficient data management systems permitting more general structuring of data than simple tree structures. Specifically, the problem of this paper may be stated in 4 parts:

(1) Find a formal analogue to a hierarchical data structure which allows cross-classification between the strata of the hierarchy, and which also has some relationship to a "tree" structure.

(2) Define an appropriate storage structure for the formal model, using previous data management formalisms as much as possible.

(3) Define the semantics of basic retrieval operations on the data structure, in the framework of a retrieval language, in such a way as to insure that the laws of a Boolean algebra are followed. This is necessary for the predictability and stability of the retrieval operations, and because it permits certain types of processing optimization.

(4) Provide in addition to the basic operational definitions the fundamental algorithms of a retrieval

package, i.e., an algorithmic definition for the elementary retrieval operations on the defined storage structure.

The paper is accordingly organized with five succeeding chapters concentrated on the above problems.

Chapter II exploits the natural hierarchy of a finite acyclic directed graph and poses this graph as a candidate for the model of a hierarchical data structure in which cross-classification is allowed.

Chapter III extends retrieval mechanisms previously defined by Lowenthal (L4) for trees, and the differences in the syntax and semantics of the retrieval expression for a more general structure are examined.

Chapter IV examines the possibility of using a more primitive retrieval language developed by Hardgrave (H2) which can both simulate the operations defined by the language in Chapter III and provide the capability of more general types of data processing. Hardgrave's language, initially developed in the context of tree structures, can easily be adapted to apply to networks.

Chapter V is involved with the definition of an abstract representation for the actual data structure, and provides a finite solution for this representation, in the form of a structure table. It is intended that this structure table be an abstract model of a real storage structure, and its significance is in its rela-

tionship to the trace table of Lowenthal (L4).

Chapter VI outlines the various basic algorithms necessary to perform retrieval operations in the data base, and is provided both as a demonstration of the efficacy of the structure table and as a basis for future programmatic experimentation.

CHAPTER II

THE ACYCLIC DATA BASE

Characterization

We seek to define a hierarchical data base which is more general than a tree-like data base, capable of fulfilling the following characteristic necessities:

(1) the representation of data partitioned by a finite set of levels $L_1,\ldots,L_k$,

(2) the representation of a labelled partition of the data entities in each level, and the characterization of each class in that partition by predefined "attributes",

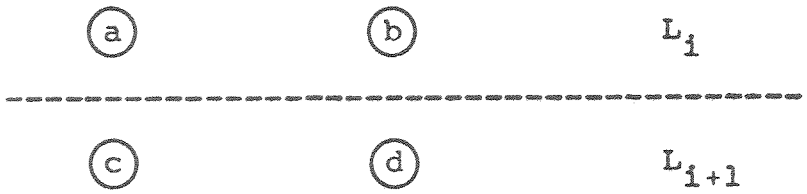
(3) the representation of hierarchical relationships between two data entities in non-adjacent levels, i.e., in the Library of Congress Subject Headings List example in Chapter I, totemism is hierarchically related to primitive religions, and these two data entities are not at adjacent levels, and

(4) the representation of exactly those hierarchical relationships of data entities which a user considers important, i.e., in the Library of Congress Subject Headings List example, a user might wish to disallow the access to totemism by way of civilization and archaeology, and allow it only by way of gods, religions,

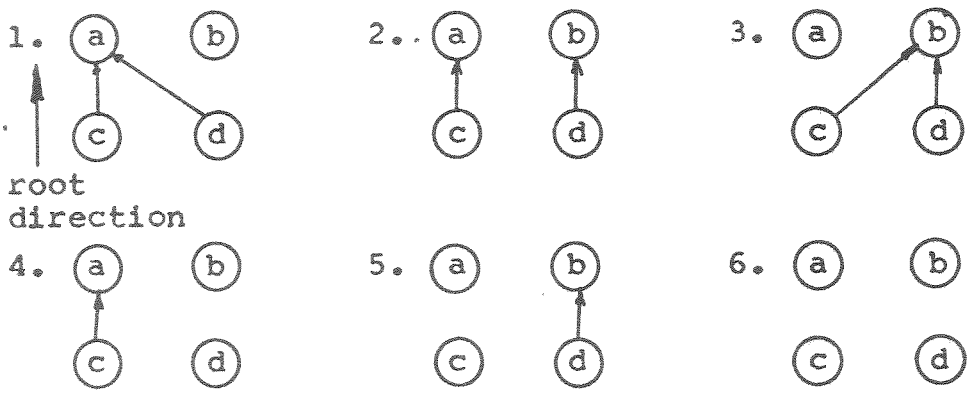
mythology, or religion, religions, mythology, or religion, primitive religion.

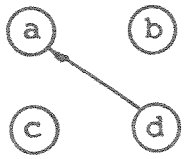
Whatever reason the user might have to exclude such an accession is not as important in the treatment of this subject as the possibility that he may have such a reason.

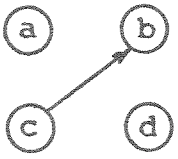
Tree-based data structures characteristically satisfy (1) and (2) above, but not properties (3) or (4). The characteristic form of a tree is that it branches in one direction and only to the adjacent level in that direction. For example, if a,b,c, and d are data entities at levels L_i and L_{i+1}, as illustrated,

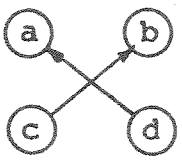


where L_i is between the "root" level of the tree and L_{i+1}, there may be only the following hierarchical relationships:

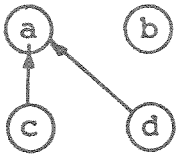


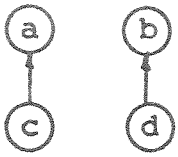
7. 

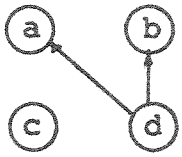
8. 

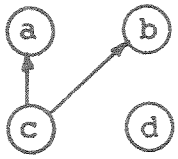
9. 

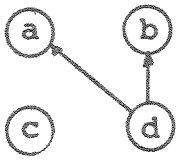
This is because the "branching" of the tree must be in only one direction, or away from the root. A data base constructed on the four characteristics noted above has the following possibilities for relating these same four entities:

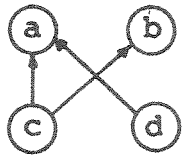
1. 

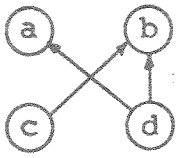
2. 

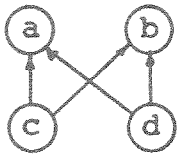
3. 

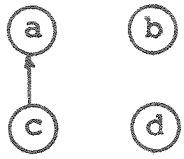
4. 

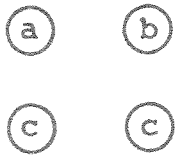
5. 

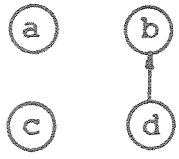
6. 

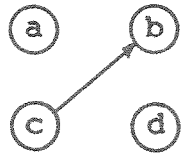
7. 

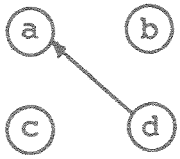
8. 

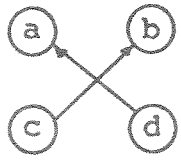
9. 

10. 

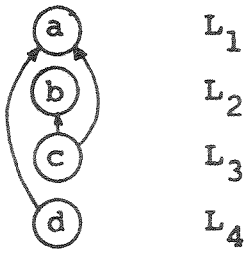
11. 

12. 

13. 

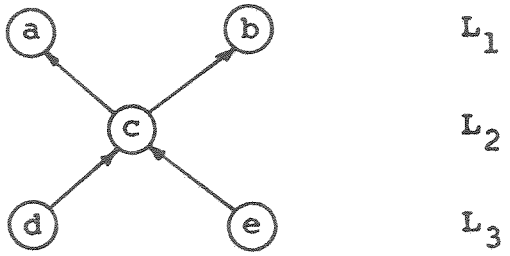
14. 

In addition, there is also the possibility from property (3) above that data entities may be hierarchically related across several levels, i.e., if a, b, c, and d are data entities at levels L_1, L_2, L_3, L_4, the following possibility exists from property (3):



Property (4) above exists because of the following typical construction:

Suppose a, b, c, d, and e are data entities; a and b at level L_1, c at L_2, and d and e at level L_3.



entity c is related to a and to b, and d and e are related to c, but d is not related to a and e is not related to b.

In order for this type of hierarchical relationship to be recorded in a data structure, it is necessary that it have characteristic (4), which requires that specific hierarchical relationships of data entities must be represented.

The significance of the four properties noted above is that they characterize the data structure of large classes of data which have a structural richness not efficiently represented in simple tree-based data structures.

It is convenient at this point to state some fundamental definitions important for subsequent discussion.

Basic Definitions

The following constructions are useful in viewing the bases for data organization and for understanding subsequent more formal discussion in this chapter:

(1) An attribute is, conceptually, the name of a piece of data. In a data management system, each attribute is identified by a unique attribute identifier. The set of all attribute identifiers in a system is finite, and is denoted by AI.

(2) A type partitioning of the set AI may be represented by the equation $AI = \bigcup_{i=1}^{m} T_i$, where each T_i, $i=1,\ldots,m$, is a set of attribute identifiers such that

if $i \neq j$ then $T_i \cap T_j = \emptyset$. In the type partitioning of AI, $\{T_1, \ldots, T_m\}$, each set T_i of attribute identifiers is ordered arbitrarily as $a_{i,1}, \ldots, a_{i,k_i}$.

(3) A finite set of value identifiers V specifies all of the values which can occur in a data base. The special value identifier "null" denotes the empty value.

(4) An attribute-value pair is an attribute identifier paired with a value identifier, and is abbreviated "a/v pair".

(5) A dataset of type T_i is a set of attribute-value pairs $\{(a_{i,1}/v_1), \ldots, (a_{i,k_i}/v_{k_i})\}$, where each $a_{i,j} \in T_i$, for $j = 1, \ldots, k_i$. Note that each dataset must contain all the attributes of its type class.

(6) A typeset of type $T_i = \{x | x \text{ is a dataset of type } T_i\}$. Since each of the attributes $a_{i,1}, \ldots, a_{i,k_i}$ is associated with the type T_i, the phrase "the type of $a_{i,j}$", where $1 \leq j \leq k_i$, refers to the symbol "T_i". The phrase "dataset x is in T_i", on the other hand, means x is an element of the typeset of type T_i.

The characteristics of a finite nodal tree (S,P,R) may be stated as follows:

i. S is a finite set of nodes, and $R \in S$

ii. P is a function such that $P: S - \{R\} \rightarrow S$

iii. For all x in S, and for Q defined as

$$Q(x) = \begin{cases} \emptyset, \text{ if } x = R \\ P(x) \cup \left\{ \bigcup_{y=P(x)} Q(y) \right\} \end{cases}$$

then $x \notin Q(x)$.

The element R is the "root" of the tree, Q(x) is the set of all ancestors of the node x, and P(x) is the set of all the ancestors of x which are directly connected to x (one level away from x toward the root, R). Because P is a function, if x is in S-R then there is only one node y in S such that y=P(x).

A <u>finite acyclic network</u> is an ordered triple (S,S_o,P) which is characterized by the following properties:

i. S is a finite set of nodes, and $S_o \subseteq S$.

ii. P is a mapping from $S-S_o$ into 2^S.

iii. For Q defined as

$$Q(x) = \begin{cases} \emptyset, \text{ if } x \in S_o \\ P(x) \cup \left\{ \bigcup_{y \in P(x)} Q(y) \right\}, \text{ otherwise} \end{cases}$$

then for all x in S, $x \notin Q(x)$.

The condition that $x \notin Q(x)$, for all $x \in S$, assures us that there can be no cycles in the graph.

This definition, which is analogous to that given for a tree above, provides for the basic algebraic structure upon which later data structures of concern in this paper will be based. Some trivial cases of the acyclic network will no doubt prove uninteresting with respect to data management application. For example, if $S_o = S$ there is no real "linkage" of any node to any other through the mapping P, and the acyclic network is

reduced to an unordered set of nodes. In the case where the mapping P maps each node into a relatively large subset of S, the network will prove not only uninteresting but also impractical for a reasonable data management data structure. If every node in S were connected to 95% of the nodes above it, for example, then there would be very little information to distinguish the various nodes from one another, and yet the structural representation of the system would be overloaded with connection information. These matters of what is "reasonable" are implementation-dependent and are beyond the scope of this paper.

An interesting characteristic of the acyclic network is that if $S_o = R$, a single node, and P is a function from $S - \{R\}$ into S, then the acyclic network is equivalent to a tree. The importance of this algebraic relationship is that it retains the possibility of a systematic simplification of certain data structures, which happen to be represented as acyclic networks, to trees, for which searching algorithms may operate on the assumption that S_o is a singleton and P is a function.

To maintain a unity between the ideas in this paper and those which have gone before, we now present a graphical analogue to the finite acyclic network, the finite acyclic directed graph.

A finite acyclic directed graph is a directed

graph, or digraph, with a finite number of nodes such that if x is a node in the digraph, there does not exist a directed path from x to x. Formally, a finite acyclic digraph is defined as follows:

Definition. A finite digraph D is an ordered pair (N,E) where N is a finite set of nodes and E is a subset of N x N. Each element of E is called an edge of D.

The notational descriptors "head" and "tail" will be used as follows:

In the diagram $(a) \xrightarrow{e} (b)$, where a and b are elements of N, and e = (a,b) is an element of E, a is said to be at the tail of e, and b is said to be at the head of e.

Definition. A cycle in a finite digraph is a finite sequence of directed edges, denoted by $(e_1,\ldots,e_k)$ such that for each i, $1 \leq i \leq k-1$, the node at the head of e_i is the node at the tail of e_{i+1}, and the node at the head of e_k is the node at the tail of e_1.

Definition. A finite acyclic digraph is a finite digraph which contains no cycles.

An equivalent of the following theorem is given in Harary (H^1) and we will not present a restatement of the proof here:

Theorem. In the finite acyclic digraph (N,E), there exists a subset N_o of N such that if $x \in N_o$ then x is not at the tail of any edge in E.

By defining, for an acyclic digraph (N,E),

$$S = N$$

$$S_o = \{ n \in S \mid \nexists\, m \in S \ni (n,m) \in E \}$$

$$P(x) = \{ m \in S \mid (x,m) \in E \}$$

it is easy to see that the acyclic digraph and the acyclic network are essentially the same. The digraph is defined in terms of sets and the network in terms of an "ancestor" mapping P.

The level concept.

With a view toward organizing the representation of the acyclic digraph for efficient data management applications, it is convenient to relate it as closely as possible to the tree. In this light, perhaps the most relevant characteristic of the tree structure is its natural separation into levels, where the level of a node x is a way of measuring its distance from the root, R, of the tree. We seek an analogous partitioning of a finite acyclic digraph in order to produce a workable storage structure later. In the finite acyclic digraph (N,E), let F be the subset of N x N such that $(a,b) \in F$ if and only if there exists $e \in E$ such that e is the edge from a to b, or $(a) \xrightarrow{e} (b)$, plus all pairs of the form (a,NIL), where $a \in N$ and a is not at the tail of any edge, and NIL denotes the second element of the pair, which is not a node in N. Algorithm A1 provides the mechanism for partitioning N into disjoint levels, denoted by L_i,

i = (1,2,...) by operating on the set F.

Algorithm A1.

A1.1. Let I be one.

A1.2. Remove all pairs from F with second element NIL.

A1.3. Collect the nodes expressed as first elements of pairs from this set, and call this <u>level</u> L_I.

A1.4. If no pairs remain in F, then stop.

A1.5. For all pairs (a,b) remaining in F, if $b \in L_I$ and there does not exist a pair (a,c) in F such that $c \in L_I$, then replace b by NIL. Otherwise delete (a,b)from F.

A1.6. Increment I by 1.

A1.7. Go to step A1.2.

The operation of this algorithm is illustrated by the example which follows.

<u>Example.</u> Let D be the directed graph illustrated by Fig. 1. We may form the set of pairs F as {(a,NIL), (b,NIL), (c,NIL), (e,a), (g,a), (e,b), (d,b), (1,c), (f,c), (o,d), (g,d), (i,f), (j,h), (m,i), (n,i), (o,j), (r,1), (p,m), (1,n), (k,n), (q,p), (r,q), (s,q), (h,e)} .

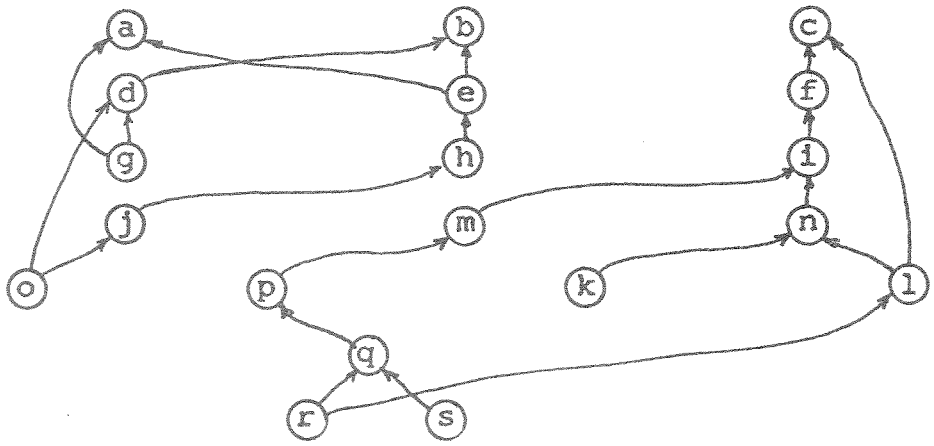


Figure 1

The first iteration of A1 produces L_1 as (a,b,c). The pair (g,a) is deleted entirely from F because $a \in L_1$ and (g,d) exists in F. After the first level, L_1, is produced, F is reduced to {(e,NIL),(,NIL),(f,NIL),(o,d), (g,d),(i,f),(j,h),(m,i),(n,i),(o,j),(r,l),(p,m),(l,n), (k,n),(q,p),(r,q),(s,q)} . Further application of A1 produces in sequence the sets

L_2 = (d,e,f)

L_3 = (g,h,i)

L_4 = (j,m,n)

L_5 = (o,p,k,l)

L_6 = (q)

L_7 = (r,s).

The hierarchy of an acyclic digraph is defined as the L-partition of levels of nodes produced by algorithm A1. Note that if a node x is directly connected by edges to several nodes $(n_1,\ldots,n_k)$, at levels $(L_{i_1}, \ldots,L_{i_k})$, respectively, then x must be in level L_m, where $m = \max(i_1,\ldots,i_k) + 1$. In the above example, node o is in level L_5, because it is adjacent to node j, which is in level L_4, even though it is also adjacent to node d, in level L_2.

The data base

The hierarchical partitioning of the acyclic digraph suggests forming a definition of a related data

structure, an acyclic data structure, which can provide

(1) a graphical representation of nodes partitioned by a finite set of levels, $L_1,\ldots,L_k$,

(2) the representation of direct hierarchical relationships (by directed edges) between nodes at any two different levels. What is lacking in the acyclic digraph (N,E) as described is the capacity to represent a partition of the nodes in a single level L_i, and the capacity to represent specific hierarchical relationships, or specific node sequences in the digraph. To provide this capacity the graph must be embellished with two additional specifications, the partition mapping P, and a set of node sequences NS.

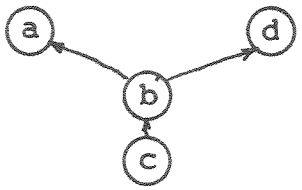
The partition mapping P maps each level L_i into a partition $L_{i,1},\ldots,L_{i,k(i)}$, where k(i) is the number of elements in the partition of level L_i, and if there are n(i) nodes in level L_i, then $1 \leq k(i) \leq n(i)$.

The set of node sequences NS, on which later definitions of path and ancestor will be based, specifies all of the hierarchical relationships among various nodes which exist. For an acyclic digraph (NE) partitioned into levels $L_1,\ldots,L_m$ by algorithm A1, the set of node sequences NS must have the following properties:

(1) If (a,b) is a subsequence of some node sequence in NS, where $a \in L_i$ and $b \in L_j$, then it must be true that $i < j$.

(2) If $(b,a) \in E$, then there exists a node sequence $S \in NS$ such that (a,b) is a subsequence of S.

We note that while it is not necessary for NS to contain E as a subset, property (2) guarantees that each simple hierarchical relationship indicated by a single directed edge in the directed graph diagram will be represented somewhere within a sequence in NS. The power gained in the specification of NS is the control of all paths longer than one directed edge. For example, in the diagram



the node sequence set $\{(a,b,c),(d,b)\}$ represents all of the edges of the graph, but eliminates the access to c through d. In other words, we may say that a is an "ancestor" of c and d is not, which suggests forming the definition of ancestry on the basis of the sequences of NS.

The embellished acyclic digraph may be expressed as an ordered quadruple D = (N,E,P,NS), where N is a finite set of nodes, E is the edge set, P is a particular partition mapping applied to the levels of D, and NS is a pre-defined set of node sequences. In order to relate

the digraph D to a data structure, we make the following correspondences between graph entities and data entities:

(1) Each node corresponds to exactly one dataset.

(2) Each element of the type partitioning of the attribute set AI corresponds to exactly one element of the partition mapping P. Since P is defined in terms of levels, we thereby associate each type and each type-set with a unique level, and thus the datasets in each level are partitioned by a set of typesets.

(3) If datasets x and y correspond to nodes a and b, respectively, the existence of the pair (a,b) as a subsequence of some sequence in NS corresponds to the existence of a hierarchical relationship of x and y, and means that x is the direct ancestor of y.

(4) The statement that x is an ancestor of y means that there exists a node sequence S in NS such that

(i) a corresponds to x,

(ii) b corresponds to y,

(iii) a and b are in S, and

(iv) a occurs to the left of b in S.

The statement that a path exists from node x to node y means x is an ancestor of y.

Two types T_1 and T_2 are said to be adjacent if some dataset of one of the types is a direct ancestor to a dataset of the other type. Typeset T_1 is ancestral to typeset T_2 if some dataset in T_1 is an ancestor of

some node in T_2.

The relationship of typesets may thus also be indicated in the form of an acyclic digraph (T,E'), where T is the set of all types, and $E' \subseteq TxT$, such that $(T_i,T_j) \in E'$ means that T_j is ancestral to T_i. For example, suppose $T = \{T_1,T_2,T_3,T_4,T_5\}$, and $E' = \{(T_2,T_1), (T_3,T_1),(T_4,T_2),(T_5,T_3),(T_5T_1),(T_5,T_2)\}$. Then we can produce the figure below.

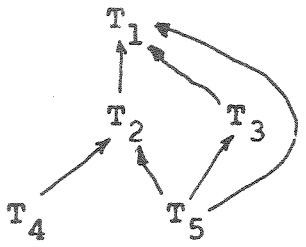


Figure 2

In the same way that node sequences are required to control hierarchies in the acyclic digraph, type symbol sequences are required to exactly specify the hierarchies among typesets. The reason is that a <u>type</u> is characterized by a set of attributes which have an inherent semantic value for the data base. It may well be unsuitable to define the ancestry of typesets on the basis of the pairs in E and assume that ancestry is transitive (i.e., to assume that ancestry is defined by the transitive closure of the pairs defining adjacency) for most specific data management applications. For example, in the above figure T_5 may be related to T_2 in a completely

different way than T_4 is related to T_2, and because of this, T_5's relationship to T_2 and T_2's relationship to T_1 may have nothing to do with each other. In general, we must suppose that such cases can occur, and design structural devices which will allow the specification of limitations if they are required, and which will also allow the complete transitivity if it truly exists in the data structure. Hence the following definitions:

$$\begin{aligned} \text{Suppose level } L_1 &= T_{1,1} \cup \cdots \cup T_{1,k(1)}, \\ \text{level } L_2 &= T_{2,1} \cup \cdots \cup T_{2,k(2)}, \\ &\vdots \\ \text{level } L_m &= T_{m,1} \cup \cdots \cup T_{m,k(m)}. \end{aligned}$$

If $T = T_{i,j}$ then LEV(T) is defined as i.

A <u>context definition</u> is a sequence of type symbols $T_{i_1,j_1} T_{i_2,j_2} \cdots T_{i_k,j_k}$ such that $i_1 < i_2 < \cdots < i_k$. In a simpler form, since each type T has a unique level, LEV(T), a context definition may be defined as a type symbol sequence $T_{i_1} T_{i_2} \cdots T_{i_k}$ where $LEV(T_{i_1}) < \cdots < LEV(T_{i_k})$.

It will be useful later to group the hierarchical information about a group of datasets according to a single context. The following definition of a <u>c-set</u> pertains to this forethought.

If $D = T_{i_1} T_{i_2} \cdots T_{i_k}$ is a context definition, then the <u>c-set</u> C associated with D is an ordered pair

(CN,CNS) where

$$CN = \bigcup_{T_{i_p} \in D} \{ n \in N \mid n \in T_{i_p} \}$$

and $CNS \subseteq NS$ (CNS is a set of node sequences) such that if $(n_1,\ldots,n_p) \in CNS$, there exists an index i_q such that $i_1 \leq i_q \leq i_k$, and

$$n_j \in T_{i_{q+j-1}}, \text{ for } j = (1,\ldots,p).$$

For example, suppose that in Figure 3, p.33, the node set N is $\{1,2,3,4,5,6,7\}$ and the set of node sequences NS is $\{(1,3,4,5),(1,3,4,6),(2,4),(2,6),(3,4,7)\}$. Suppose context definition $D = T_2T_3T_5$. The c-set C associated with D is the subset of N contained in the typesets T_2, T_3, T_5, and the sequences in NS which conform to the type sequence $T_2T_3T_5$. Thus

$$C = \{\{3,4,7\}, \{(3,4,7)\}\} .$$

Similarly for context definition $D = T_0T_2T_3T_4$, the c-set $C = \{\{1,3,4,5,6\}, \{(1,3,4,5),(1,3,4,6)\}\}$.

The <u>context definition</u> bears a close relationship to the <u>type family</u> of Lowenthal (L4), and correspondingly, a c-set bears a close relationship to a <u>family</u> as defined by Lowenthal. Briefly, Lowenthal defines a

type family, for a tree structure as a list of type symbols $T_0T_{i_1}T_{i_2}\ldots$, somewhat similarly to a context definition. The differences are as follows:

(1) T_0 is a member of every type family, and is the unique "top" node's type. (T_0 is the "root" of the tree.)

However, in the acyclic digraph-based structure, all of the context definitions will not, in general, contain a common type symbol. Context definitions may begin and end at any level, and may have no common type symbols at all.

(2) If $T_0T_{i_1}\cdots T_{i_k}$ and $T_0T_{j_1}\cdots T_{j_p}$ are two different type families, then they can only be different in the following sense:

There must exist an integer n such that $1<n<\min(k,p)$ and such that if $m\leq n$ then $T_{i_m}=T_{j_m}$ and if $m>n$ then $T_{i_m}\neq T_{j_m}$,

whereas two context definitions may differ in only one type symbol. In order for two context definitions $T_{i_1}\cdots T_{i_k}$ and $T_{j_1}\cdots T_{j_p}$ to be the same, it must be true that $k = p$ and $i_n = j_n$, for $n = (1,\ldots,k)$.

(3) Two adjacent type symbols in a type family must be at adjacent levels. In a context definition, this is not a criterion. The only requirement is that the levels of the types be increasing to the right, i.e.,

if $T_x T_y$ appears in a context definition, then the level of T_x is less than the level of T_y. In a type family, the level of T_y would be the level of T_x plus one.

For this paper, an acyclic data base consists of two basic parts, (1) a structural definition, which defines each type in terms of its attributes, and the contexts, or type sequences, in the data base, and (2) a representation, which records the node sequences specified by NS, in a more efficient manner. It is thus necessary in a structural definition to specify (a) the attributes comprising each type, and (b) the context definitions, which give information as to the type sequences or the type structure.

Example.

The following discussion is based on the small data base diagrammed in Fig. 3, where $N = \{1,2,3,4,5,6,7\}$, $E = \{(3,1),(4,3),(4,2),(5,4),(6,4),(6,2),(7,4)\}$, and $NS = \{(1,3,4,5),(1,3,4,6),(2,4),(3,4,7),(2,6)\}$. Applying algorithm A1 to (N,E) produces the partition $L_1 = \{1,2\}$. $L_2 = \{3\}$, $L_3 = \{4\}$, $L_4 = \{5,6,7\}$. We wish to partition each level by type sets, where $T_0 = \{1\}$, $T_1 = \{2\}$, $T_2 = \{3\}$, $T_3 = \{4\}$, $T_4 = \{5,6\}$, $T_5 = \{7\}$, and the types are characterized by attributes as follows:

T_0: GROUP

T_1: FED. TAX REVIEW

T_2: STATE

T_3: CITY

T_4: DISTRICT

D.A.

T_5: PARK

MANAGER

A diagram of the data base as an acyclic digraph appears in Fig. 3, with a data value supplied for each attribute. Node 2 is a reflection of a decision to ignore state boundaries in designing a tax review area and both the city Pampa and the district number 2 in Pampa are related to node 2. Several relationships exist in the data base as depicted which are undesirable for a retrieval operation. Let us suppose that the type of each node is depicted by the type symbol in Fig. 2, and that the data base designer has specified a structural definition which allows the following contexts:

$T_0T_2T_3T_4$

T_1T_3

$T_2T_3T_5$

T_1T_4.

These are diagrammed as follows:

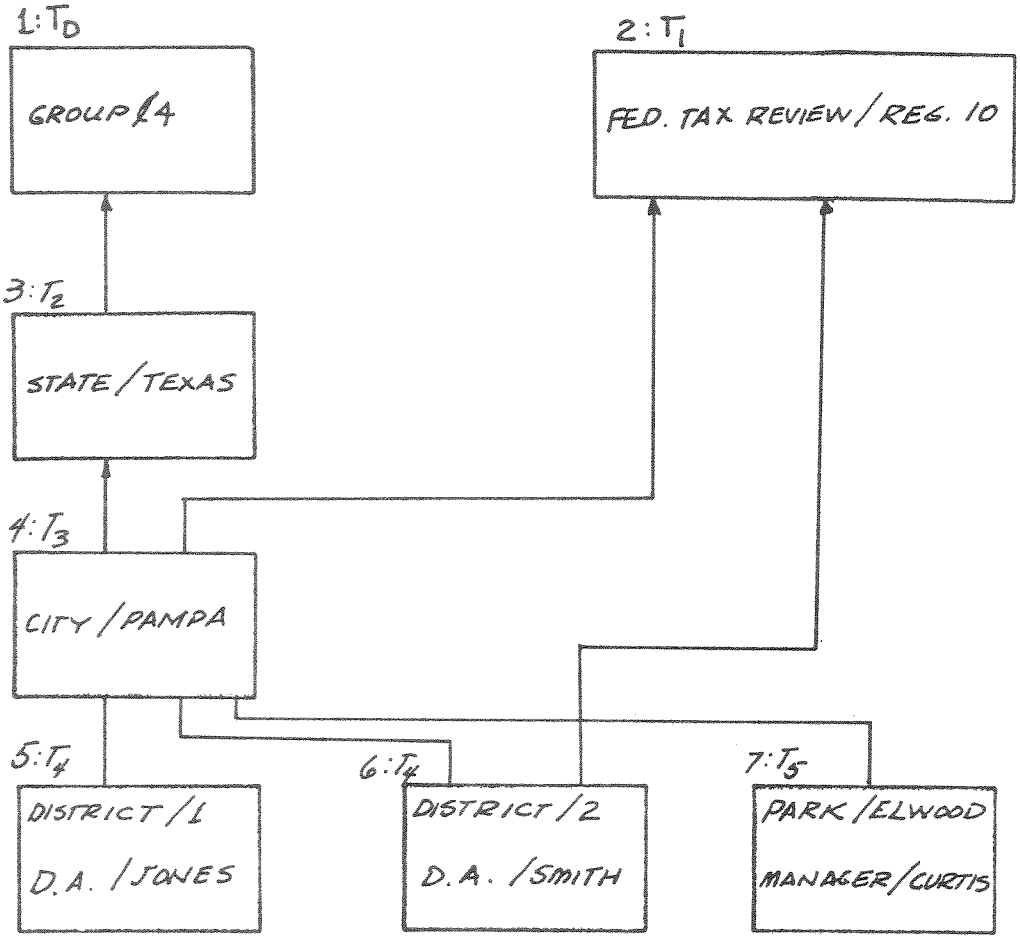

1: T_D
GROUP 14
2: T_1
FED. TAX REVIEW / REG. 10
3: T_2
STATE / TEXAS
4: T_3
CITY / PAMPA
5: T_4
DISTRICT / 1
D.A. / JONES
6: T_4
DISTRICT / 2
D.A. / SMITH
7: T_5
PARK / ELWOOD
MANAGER / CURTIS

Figure 3

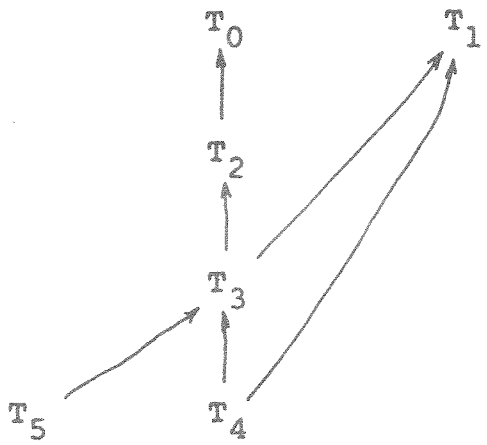
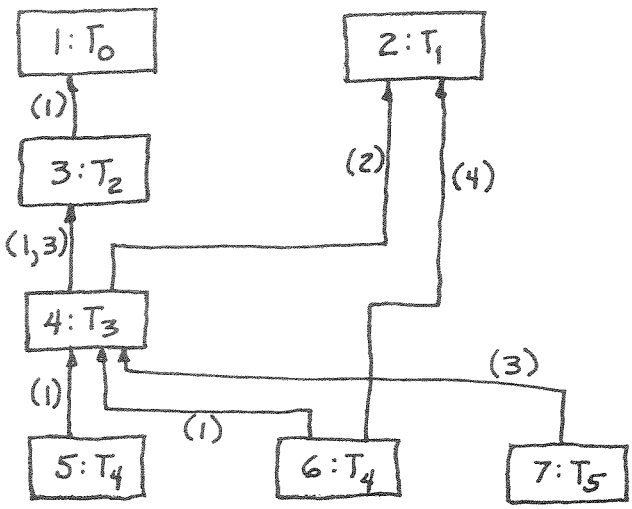


Figure 4

A problem in structural over-richness that appears in the acyclic digraph is that the context $T_1T_3T_5$ appears to exist in the data structure in Figure 3 but does not occur in the structural definition. This is also true of the sequences $T_1T_3T_4$ and $T_0T_2T_3T_5$. However, if we embellish the acyclic digraph with a simple edge-label scheme, the problem can be solved.

First, let us number the contexts specified as $1{:}T_0T_2T_3T_4$, $2{:}T_1T_3$, $3{:}T_2T_3T_5$, and $4{:}T_1T_4$. Then we label an edge with the number of the context if it belongs in that context. Figure 5 shows the structure with edge labels.



Acyclic Digraph with Edge Labels

Figure 5

The augmentation of the data structure with edge labels, in this example, removes the possibility of finding spurious relationships due to unspecified type adjacencies. Thus, the digraph so labelled corresponds to a data base defined with structural context specification.

A more elaborate edge label could be applied to each edge to record the information about node sequences on the acyclic digraph. The node sequences would be labelled, like the context definitions, and if an edge was a part of a particular node sequence it would carry the label of that sequence. Because of the complexity of this sort of scheme and because it is not useful as a data

management tool, it will not be developed. It is worth mentioning as a representation of the node sequence set NS, however, because there do exist general graph-processing software systems in which small prototypes of these data structures could be implemented for algorithmic design research.

The following more comprehensive example of an acyclic data base is related to current research in the field of data management, and shows how a tree-structured data base may be imbedded within an acyclic data base. Its importance grows out of a great deal of debate revolving about the semantics of Boolean operations on structured data bases, centered about this example. It provides the necessary stereotype structures for a tree, and the opportunity for expansion into an acyclic data base.

Sample Data Base

We wish to demonstrate the relationship of the acyclic data base to a use of the tree structure proposal by Lowenthal (L4). Lowenthal's sample data base is reproduced in Figures 6 and 7.

Types and Associated Attributes

T_1: States

$A_{1,1}$: Name of State

$A_{1,2}$: Population of State

T_2: Congressional Districts in States

$A_{2,1}$: Population of District

$A_{2,2}$: Name of Congressman

T_3: Voting Record of Congressman

$A_{3,1}$: House Bill Number

$A_{3,2}$: Vote

T_4: Financial Involvement of Congressman

$A_{4,1}$: Name of Company

T_5: Major Cities in State

$A_{5,1}$: Name of City

$A_{5,2}$: Population of City

$A_{5,3}$: Growth Rate

Hierarchical Relationships Among Types

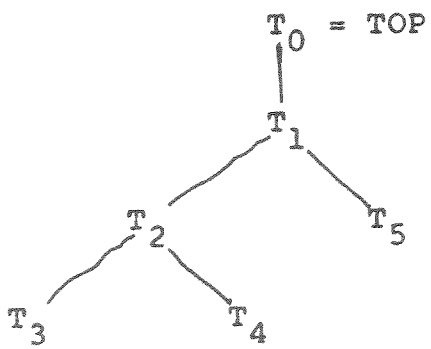
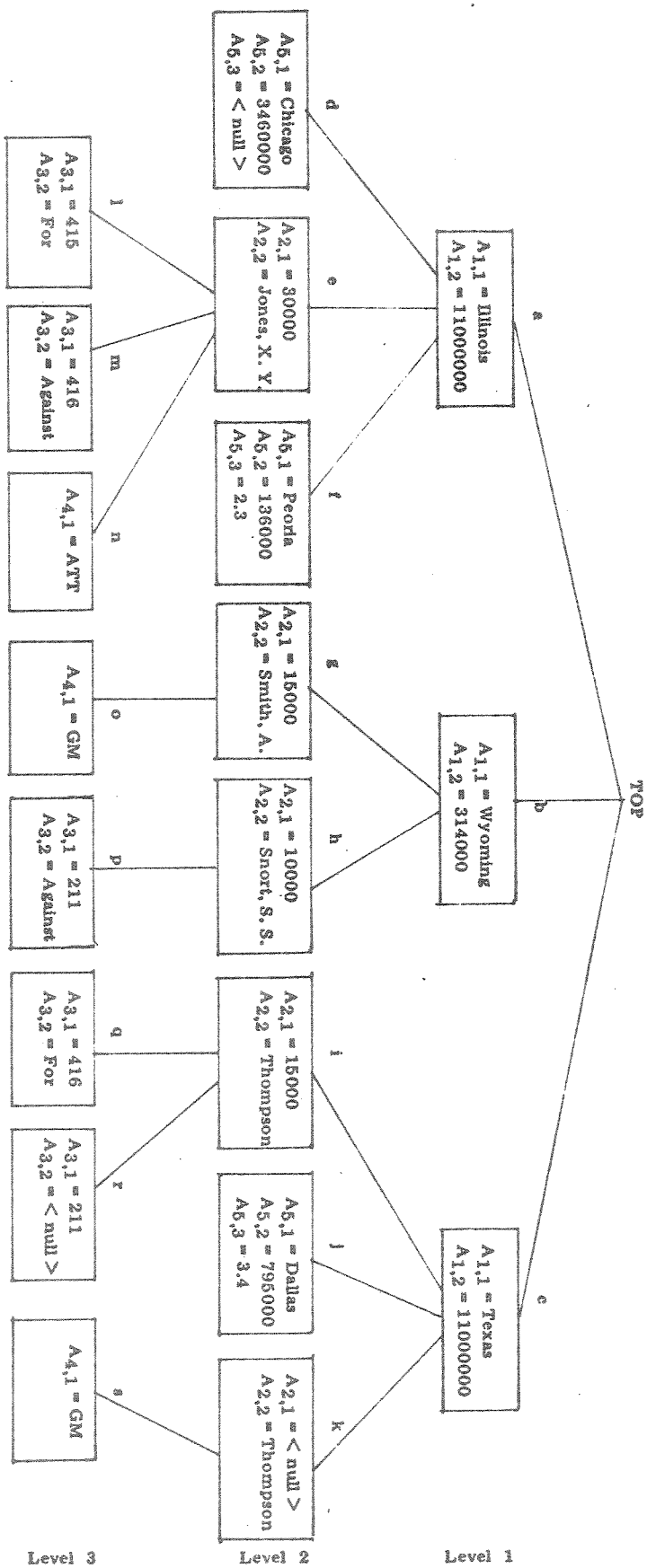


Figure 6



$T_1 = \{a,b,c\}$ $T_2 = \{e,g,h,i,k\}$ $T_3 = \{l,m,p,q,r\}$ $T_4 = \{n,o,s\}$ $T_5 = \{d,f,j\}$

Figure 7

Lowenthal allows several "families" of types here, T_1,T_2,T_3, T_1,T_2,T_4, and T_1,T_5. In this example and in the standard "tree-based" data structure, if a "family" is implied by the type structure, then it is assumed to have some meaning in the data structure. That is to say, the contexts of the data base on which retrieval may be performed are implicit in the data base definition and are not explicitly stated as families. We will soon see that by requiring an explicit definition of contexts we may produce a freer interconnection of nodes and eliminate spurious relationships between nodes and types which may arise from this freedom.

It is interesting to make an expansion of Lowenthal's example to produce an acyclic network. Let us suppose that for each company in the data base, there are a number of states in which the company has headquarters, and within each state, we wish to store information about the companies' headquarters for each city. Thus, the nodes in type T_1, the "state" nodes, become candidates for repeating groups, or repeating datasets, for each node in type T_4, and the location of company headquarters becomes a repeating group for each state. Let us suppose further that each Congressman's vote has some relationship to a particular group of company headquarters. We may provide this relationship and the company headquarters' repeating groups at the same time by adding a set

of "city" nodes at one level less than the state nodes, allowing connections from "company" nodes to "state" nodes (this crosses two levels), and connections from "vote" nodes to "city" nodes. This is diagrammed in Figure 8.

Node labels, shown as lower case letters above the nodes, represent unique addresses of datasets, in the case of the acyclic network. In Lowenthal's example, they represent node addresses, which may not be the same as dataset addresses, in the way Lowenthal defines his tree data base. For the acyclic network, however, nodes and datasets are in 1:1 correspondence.

Sample Data Base Definition

(A) Types and Associated Attributes

T_1: Company Headquarters

- $A_{1,1}$: Name of City
- $A_{1,2}$: Address of Headquarters
- $A_{1,3}$: Phone Number

T_2: States

- $A_{2,1}$: Name of State
- $A_{2,2}$: Population of State

T_3: Congressional Districts

- $A_{3,1}$: District Number
- $A_{3,2}$: Population
- $A_{3,3}$: Congressman

T_4: Major Cities

- $A_{4,1}$: Name of City
- $A_{4,2}$: Population
- $A_{4,3}$: Growth Rate

T_5: Voting Records

- $A_{5,1}$: House Bill Number
- $A_{5,2}$: Vote

T_6: Financial Involvement of Congressman

- $A_{6,1}$: Name of Company

(B) Context Definitions

C_1: T_1T_5

C_2: T_2T_4

C_3: $T_1T_2T_6$

C_4: $T_2T_3T_5$

C_5: $T_2T_3T_6$

Graph of Type Structure

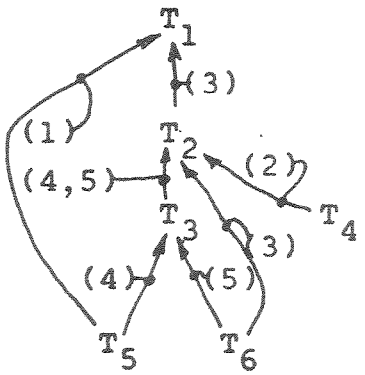
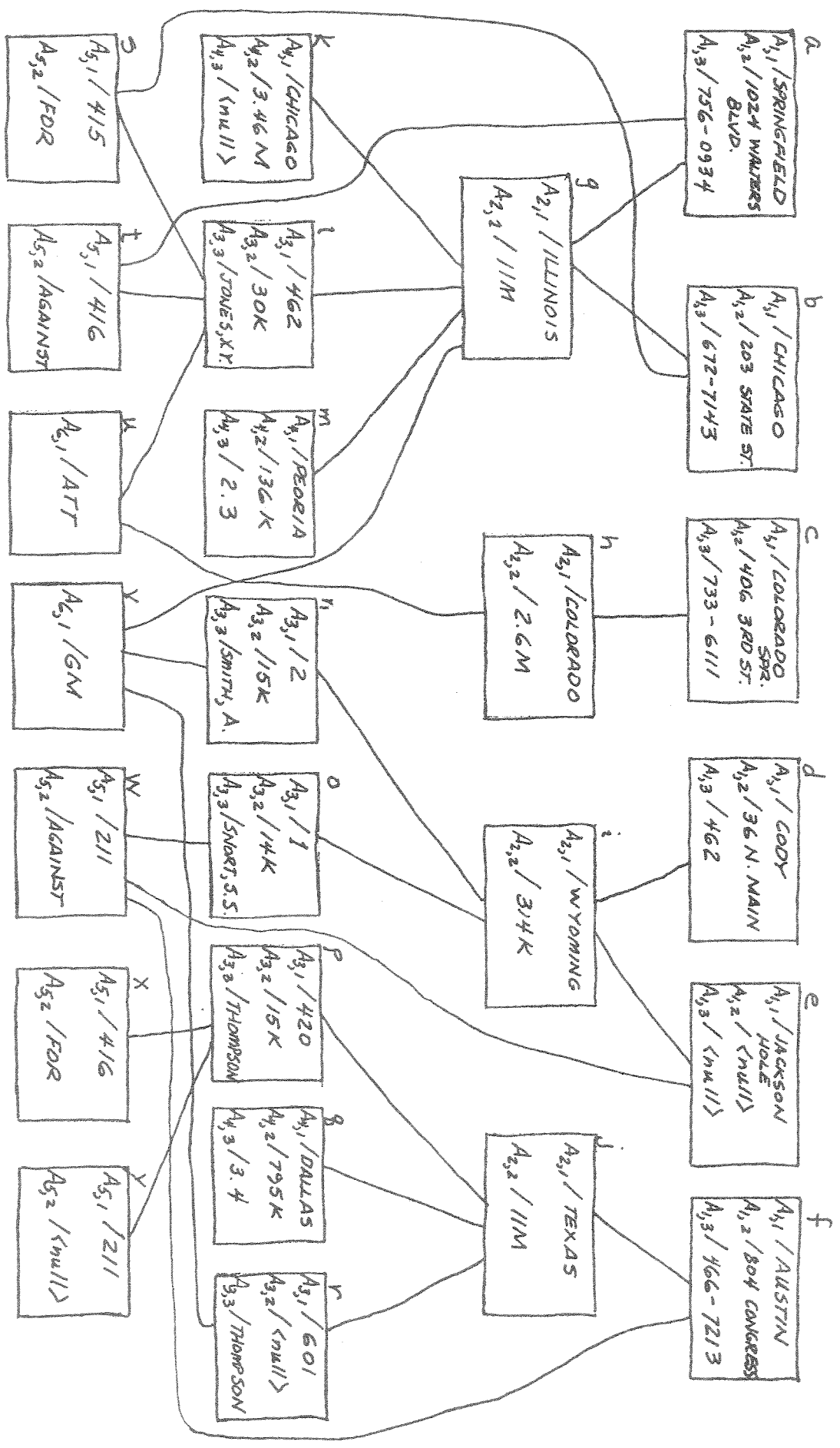


Figure 8
(continued on next page)



T_1: {a,b,c,d,e,f}
T_2: {g,h,i,j}
T_3: {l,n,o,p,r}

T_4: {k,m,q}
T_5: {s,t,w,x,y}
T_6: {u,v}

Figure 8 (continued)

Node Sequences (specified by some data input language and procedure)

NS = {(a,t),(b,s),(e,w),(f,w),(g,k),(g,m),(j,q),(a,g),
(b,g,v),(c,h,u),(d,i),(e,i),(f,j),(g,l,s),(g,l,t)
(i,n),(i,o,w),(j,p,x),(j,p,y),(j,r),(g,l,u),(i,n,v),
(j,r,v)}

Figure 8 (continued)

Explanation of Figure 8.

(1) The structure has four levels, and six types. Note that nodal (dataset) redundancy is not present. By this statement we mean that only unique nodes are represented in the digraph so that, for example, the node where $A_{6,1}$ is paired with GM is connected to two nodes in the level directly above, rather than being duplicated as in Lowenthal's tree.

(2) In the context definitions, we do not allow transitivity in the sense that the existence of context definitions $T_1T_2T_6$ and T_2T_4 does not imply the existence of a context definition $T_1T_2T_4T_6$. The context definitions are the exact specification of the allowable paths among nodes of various types.

(3) The node sequences are specified by some data input process. These represent certain "paths" through the network. Notice the omission of particular paths or possible node sequences, such as (a,g,v), even though (a,g) and (b,g,v) are elements of NS. The drawing of the acyclic digraph thus is somewhat misleading about the nodal hierarchies when the edges are left unlabelled.

(4) The c-sets for context definitions C_1 through C_5 may be listed as follows:

$$C_1: \quad \{\{a,b,c,d,e,f,s,t,w,x\}, \{(a,t),(b,s),(e,w),(f,w)\}\}$$

C_2: $\{\{g,h,i,j,k,m,q\}, \{(g,k),(g,m),(j,q)\}\}$

C_3: $\{\{a,b,c,d,e,f,g,h,i,j,u,v\}, \{(a,g),(b,g,v),(c,h,u),(d,i),(e,i),(f,j)\}\}$

C_4: $\{\{g,h,i,j,l,n,o,p,r,s,t,w,x,y\}, \{(g,l,s),(g,l,t),(i,n),(i,o,w),(j,p,x),(j,p,y),(j,r)\}\}$

C_5: $\{\{g,h,i,j,l,n,o,p,r,u,v\}, \{(g,l,u),(i,n,v),(j,r,v)\}\}$

There is much redundancy in these c-sets. One of the problems in suggesting a form for the storage structure to represent c-sets is to manage this redundancy in an efficient way.

Summary

In summary, the acyclic digraph has two properties commensurate with the classes of data structures introduced in Chapter I. It provides connections across several levels, and provides for the connection of a node at any particular level to many nodes at many different levels, both lower and higher. These qualities are ideal for data which is organized into hierarchies and in which cross-classification and multiple classification can occur. It is clear that the problems with an unlabelled acyclic digraph are mostly a result of its richness. The specification of contexts when defining

a data structure removes the possibility of making improper associations between classes of datasets in various types. The problem of efficiently recording exactly those node sequences which the user specifies will be considered in Chapter IV, with the development of a storage structure concept.

To gain a clear understanding of how the storage structure is to be used, it is necessary first to describe the syntax and semantics of the retrieval expression. This is the major topic in Chapter III.

CHAPTER III

RETRIEVAL WITH A NON-PROCEDURAL LANGUAGE

The two basic approaches to the retrieval of data from the acyclic data base will be the procedural language approach, and the non-procedural language approach. The non-procedural language defined in this chapter stems from previous data management systems which were based on tree structures. The advantage of the non-procedural language is that it is a high-level language using English-like imperatives to demand various sorts of retrievals of the system. It is thus designed for users with little or no knowledge of programming languages or formal data structures. Various semantic problems with the non-procedural language can cause difficulties with users, and they can be overcome by building up the user's knowledge of the data structure. Since this somewhat defeats the purpose of the non-procedural language, there is room for development of a procedural language, which naturally requires more in the way of programming knowledge, but which operates at a more primitive level and is thus less fraught with semantic difficulties. A recent development by Hardgrave (H2) is used in

Chapter IV to show how a procedural language could be applied to networks.

In exploring the possibilities of a non-procedural retrieval language the three major problems are as follows:

(1) to define the syntax of a non-procedural retrieval expression,

(2) to give rules which govern the formation of the semantically valid retrieval expression, and

(3) to define the result, or meaning of processing a retrieval expression for the acyclic data base.

In presenting solutions to these three problems, several key ideas maintain the continuity of the research in this area with that performed mainly in the context of tree-based data structures.

Lowenthal (L4) formalizes the retrieval expression by considering Boolean operations on simple retrieval conditions. The development of a tree algebra for the data structure provides for Boolean combinations of sets of nodes at different levels. Hardgrave (H2) introduces a broader retrieval algebra for tree structures which gives the user much more control over the method in which sets of nodes at different levels are combined in Boolean operations. S.Y. Bang, in a recent study (B2), characterizes the problems of retrieval from data structures

in terms of showing that certain essential operations are non-homomorphic.

The work of Bang mentioned is algebraically fundamental, and thus applies nicely to the acyclic data base. Lowenthal's tree algebra shows the need for certain kinds of operations in the data structure. Hardgrave's results lend a measure of strength to the retrieval method stated here, for he finds it expedient to define a fairly similar process for trees, again allowing the user to control the manner in which sets of nodes are combined (see Ch. IV).

The Retrieval Expression

In rough terms, a retrieval expression is a means of (1) specifying a certain condition which the user feels is adequate to identify those datasets he is interested in, and (2) ordering a retrieval processor to access the data structure and produce those datasets qualified.

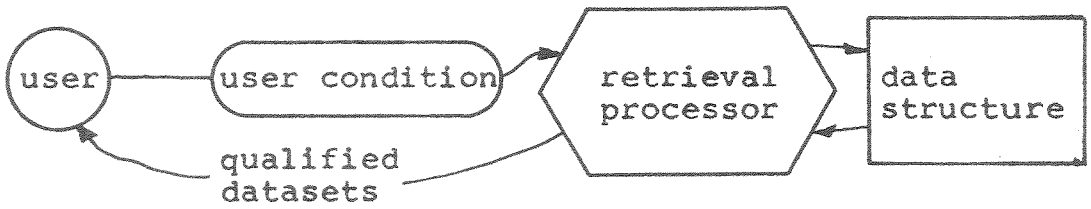


Figure 9

The information-flow diagram above summarizes this rough idea.

The general form of a retrieval expression, or a request to the data management retrieval processor to extract some explicit information from the data base is

PRINT ⟨A⟩ WHERE ⟨qc⟩

where ⟨A⟩ is a set of attributes of a single type, and ⟨qc⟩ is a qualifying clause.

The information extracted will be the values of the attributes listed in A which are in datasets hierarchically related to the nodes qualified by the qualifying clause ⟨qc⟩ . Thus the final operation implied by the retrieval expression is a projection (to be defined later) of the qualified set to the type of the attributes in A. The idea that this projection should be included in a formal definition of retrieval is attributed to S.Y. Bang (B2).

The Qualifying Clause

Since data sets are made up of attribute/value pairs, the simplest retrieval expression would contain a qualifying clause which consists of an attribute symbol, a relational symbol and a value. This simplest qualifying clause is called a simple condition and its form may be expressed by the symbol aRv (attribute-relation symbol-value). Since the attribute a may appear in only one

type, the type of aRv is uniquely defined as the type of a. Although relational symbols are normally taken from the set $\{=, \neq, <, \leq, >, \geq\}$, we will simply assume the existence of a finite relational symbol set RS and let the symbol R denote a member of RS.

Definition of the predicate SAT (aRv,n)

Suppose that n is a dataset of type T, and the type of aRv is T, and n contains the attribute/value pair a/v'. We define the predicate SAT(aRv,n) as follows:

$$SAT(aRv,n) = \begin{cases} 1, \text{ if v' bears the relationship R to v} \\ 0, \text{ otherwise} \end{cases}$$

If SAT(aRv,n) = 1 the dataset n is said to satisfy the simple condition aRv.

With the use of the predicate SAT, we may define the result of applying a simple condition against an entire typeset.

Definition of SEL

If S is a simple condition of the form aRv, where a is of type T, then the selection function SEL for S in T is defined as

$$SEL(S,T) = \{ n \in T \mid SAT(S,n) = 1 \} .$$

Syntax of the qualifying clause

The syntax of the qualifying clause, where <a> denotes an attribute, <R> denotes a member of the relational set RS, <v> denotes a value, A denotes a set of attributes of the same type, and <C> denotes a context label, is given below.

<simple condition> ::= <a> <R> <v>

<selection clause> ::= <simple condition> | (<a> HAS <qualifying clause>) |
(<a> HAS <qualifying clause> IN <C>) |
(<selection clause>) |
(<selection clause> or <selection clause>) |
(<selection clause> and <selection clause>) |
not <selection clause>

<qualifying clause> ::= <selection clause> | (<qualifying clause>) |
(<qualifying clause> OR <qualifying clause>) |
(<qualifying clause> AND <qualifying clause>) |
NOT <qualifying clause>

<retrieval expression> ::= PRINT <A> WHERE <qualifying clause> |
PRINT <A> WHERE <qualifying clause> IN <C>

Semantics of the retrieval expression

The fairly orthodox syntax on the previous page is sufficient to provide Boolean combinations of simple conditions. The use of two sets of Boolean operators {and, or, not}, and {AND, OR, NOT} as suggested by Bang (B2) arises from difficulties in processing the qualifying clause to produce the set of nodes said to be "qualified". A discussion of Bang's result appears in Appendix A.

The following semantic rules control the formation of the retrieval expression:

S1. The type of a simple condition <a> < R > <v> is the type of the attribute a.

S2. If a selection clause is a simple condition, the type of the selection clause is the type of the simple condition.

S3. If a selection clause is of the form ($\langle sc_1 \rangle$ or $\langle sc_2 \rangle$),($\langle sc_1 \rangle$ and $\langle sc_2 \rangle$), not $\langle sc_1 \rangle$, where sc_1 and sc_2 are selection clauses, then the type of sc_1 must be the type of sc_2 and this is the type of the selection clause.

S4. If a selection clause is of the form <a> HAS <qc> IN <C>, then the type of the selection clause is the type of the attribute a.

S5. If a selection clause is of the form <a> HAS <qc>, then the type of the selection clause is the

type of attribute a.

S6. In a retrieval expression, the image type T_A is the type of the attributes listed in the set A.

S7. If a retrieval expression is of the form

PRINT <A> WHERE <qc>,

(<qc> is a qualifying clause)

the type T_A and the types of the selection clauses making up <qc> must all appear in the same context definition. If these types all appear in more than one context definition, they do not uniquely determine a context and the retrieval expression is ambiguous and therefore illegal. If the image type T_A and the types of the selection clauses making up <qc> do not all appear in a single context definition, then the retrieval expression is illegal.

S8. If a retrieval expression is of the form

PRINT <A> WHERE <qc> IN <C>,

the type T_A and the types of the selection clauses making up <qc> must all appear in the context definition C.

S9. If a selection clause is of the form

<a> HAS <qc>,

then the type of attribute a and the types of the selection clauses making up <qc> must all appear in the same context definition. If these types all appear in more than one context definition, they do not uniquely determine a context and the selection clause is ambiguous. If the type of attribute a and the types of the selection clauses making up <qc> do not all appear in a single context definition, then the selection clause is illegal.

S10. If a selection clause is of the form

<a> HAS <qc> IN <C>,

where C is a context label, then the type of attribute a and the types of the selection clauses making up <qc> must all appear in the context definition C.

The Retrieval Mapping

For the results of retrieval expressions, we define a retrieval mapping F, from the set of all retrieval expressions which are semantically correct, to the power set of the set of all nodes, or 2^N.

In order to define F, we have need of a projection mapping P, defined as follows:

Suppose that D is an acyclic data base corresponding to the acyclic digraph (N,E,P,NS), and C is a

context definition $T_{i_1} \ldots T_{i_k}$, $T = T_{i_j}$, for some j, $1 \leq j \leq k$, and $s \subseteq T_{i_m}$, where $1 \leq m \leq k$. Then the projection P of set S into T is defined in three cases, as follows:

i. if $m = j$, then $P(T,C,S) = S$

ii. if $m > j$, then $P(T,C,S) = \{ x \in T \mid \exists (n_1,\ldots,n_{m-j}) \in NS$ such that ($\exists$ p,q, where $1 \leq p < q \leq m-j$, such that $x = n_p, n_q \in S$), and $n_k \in T_{i_{j+k-1}}$, $k=1,\ldots,m-j\}$

iii. if m j, then $P(T,C,S) = \{ x \in T \mid \exists (n_1,\ldots,n_{j-m}) \in NS$ such that ($\exists$ p,q where $1 \leq p < q \leq j-m$, such that $x = n_q$, $n_p \in S$), and $n_k \in T_{i_{m+k-1}}$, $k=1,\ldots,j-m\}$

where NS is, of course, the set of node sequences.

The projection mapping P thus utilizes those node sequences which conform to certain type sequences, or contexts.

The retrieval mapping F may now be defined on the various syntactic forms of the retrieval expression grammar.

F1. If S is a simple condition of type T, then
F(S) = SEL(S,T)

F2. If <a> HAS <qc> is a selection clause where attribute a is of type T, and C is a context uniquely determined by T and the types of the selection clauses making

up <qc>, then

$$F(\langle a\rangle \text{ HAS } \langle qc\rangle) = P(T,C,F(\langle qc\rangle))$$

F3. If (<a> HAS <qc> IN <C>) is a selection clause where a is of type T, and the clause is valid by rule S10, then

$$F(\langle a\rangle \text{ HAS } \langle qc\rangle \text{ IN } \langle C\rangle) = P(T,C,F(\langle qc\rangle))$$

F4. If S is a selection clause, then

$$F((S)) = F(S)$$

F5. If S_1 and S_2 are selection clauses of type T, then

$$F(S_1 \text{ or } S_2) = F(S_1) \cup F(S_2), \text{ and}$$
$$F(S_1 \text{ and } S_2) = F(S_1) \cap F(S_2)$$

F6. If S is a selection clause of type T, then

$$F(\text{not } S) = \{x \mid x \text{ is of type } T\} - S.$$

For the following definitions, if Q is a qualifying clause preceded by HAS, then T' denotes the type of the attribute preceding the HAS operator. If Q is not

preceded by HAS, then T' denotes the type of the attribute set A, or T_A.

F7. If Q is a qualifying clause, then

$$F((Q)) = F(Q)$$

F8. If Q_1 and Q_2 are qualifying clauses of types T_1 and T_2, respectively, and the context C is uniquely determined by T' and the types of the selection clauses making up Q_1 and Q_2, then if Q is of the form Q_1 OR Q_2, then

$$F(Q) = F(Q_1 \text{ OR } Q_2) = P(T',C,F(Q_1)) \cup P(T',C,F(Q_2)),$$

and if Q is of the form Q_1 AND Q_2, then

$$F(Q) = F(Q_1 \text{ AND } Q_2) = P(T',C,F(Q_1)) \cap P(T',C,F(Q_2))$$

F9. If Q is a qualifying clause composed of selection clauses of types $T_{i_1},\ldots,T_{i_k}$, and C is the context which is uniquely determined by T' and $T_{i_1},\ldots,T_{i_k}$, then NOT implies complementation in the universe U defined as

$$U = \bigcup_{j=1}^{k} (P(T',C,T_{i_j})), \text{ and}$$

$$F(\text{NOT } Q) = U - P(T', C, F(Q))$$

F10. If RE is a retrieval expression, the attribute set A is of type T, and the context C is uniquely determined by the type T_A and the types of the selection clauses making up $\langle qc \rangle$, then

$$F(RE) = P(T_A, C, F(\langle qc \rangle))$$

Boolean Algebras

All operations of retrieval produced by the selection clause occur as set operations within a single type. If the type is T, then, all selection operations in T produce members of the power set 2^T, which together with the set operators $\cup$, $\cap$, -, form an archetypal Boolean algebra. The importance of this, of course, is to enable the reduction of selection clauses to elementary forms, such as disjunctive normal form, and to maintain the predictability of which datasets will be selected by any given selection clause.

Similarly, the qualifying clause produces set operations in the set of nodes of type T', whether T' be T_A or the type of an attribute preceding the HAS operator. The universe for these operations is U as defined, so that, for any given qualifying clause, the set operations and 2^U form a Boolean algebra.

Examples of Selection and Qualification Clauses

Several examples of selection and qualification clauses will help to illustrate these ideas. These expressions are based on the sample data base depicted in Figure 8.

E1. PRINT $A_{2,1}$ WHERE $A_{4,1}$ EQ PEORIA.

1. $T_A = T_2$
2. Types T_2 and T_4 occur together only in context C_2, so C_2 is uniquely determined by these types.
3. F(PRINT $A_{2,1}$ WHERE $A_{4,1}$ EQ PEORIA)
   $=P(T_2,C_2,F(A_{4,1}\ \text{EQ PEORIA}))$
   $=P(T_2,C_2,\{m\}) = \{g\}$

E2. PRINT $A_{3,1}$ WHERE ($A_{5,1}$ EQ 415 and $A_{5,2}$ EQ AGAINST)

1. $T_A = T_3$
2. Types T_3 and T_5 uniquely determine the context C_4.
3. F(PRINT $A_{3,1}$ WHERE ($A_{5,1}$ EQ 415 and $A_{5,2}$ EQ AGAINST)
   $=P(T_3,C_4,F(A_{5,1}\ \text{EQ 415 and}\ A_{5,2}\ \text{EQ AGAINST})$
   $=P(T_3,C_4,(F(A_{5,1}\ \text{EQ 415}) \cap F(A_{5,2}\ \text{EQ AGAINST})))$
   $=P(T_3,C_4,(\{s\} \cap \{t,w\}))$
   $=P(T_3,C_4,\ \emptyset\ ) = \emptyset$

E3. PRINT $A_{3,1}$ WHERE ($A_{5,1}$ EQ 415 AND $A_{5,2}$ EQ AGAINST)

1. $T_A = T_3$
2. The types T_3 and T_5 uniquely determine the context C_4.
3. $F(\text{PRINT } A_{3,1} \text{ WHERE } (A_{5,1} \text{ EQ 415 AND } A_{5,2} \text{ EQ AGAINST})$
   $= P(T_3, C_4, F(A_{5,1} \text{ EQ 415 AND } A_{5,2} \text{ EQ AGAINST})$
   $= P(T_3, C_4, P(T_3, C_4, F(A_{5,1} \text{ EQ 415})) \cap P(T_3, C_4, F(A_{5,2} \text{ EQ AGAINST})))$
   $= P(T_3, C_4, P(T_3, C_4, \{s\}) \cap P(T_3, C_4, \{t,w\})$
   $= P(T_3, C_4, \{1\} \cap \{1,o\}) = P(T_3, C_4, \{1\}) = \{1\}$

Examples E2 and E3 point out the difference between the two operators and and AND.

E4. PRINT $A_{2,1}$ WHERE ((not($A_{6,1}$ EQ GM)) OR $A_{1,1}$ EQ AUSTIN)

1. $T_A = T_2$
2. Types T_1, T_2, and T_6 uniquely determine context C_3.
3. $F(\text{PRINT } A_{2,1} \text{ WHERE } ((\text{not}(A_{6,1} \text{ EQ GM})) \text{ OR } A_{1,1} \text{ EQ AUSTIN}))$
   $= P(T_2, C_3, F((\text{not}(A_{6,1} \text{ EQ GM})) \text{ OR } A_{1,1} \text{ EQ AUSTIN}))$
   $= P(T_2, C_3, P(T_2, C_3, (F(\text{not}(A_{6,1} \text{ EQ GM})))) \cup P(T_2, C_3, (F(A_{1,i} \text{ EQ AUSTIN}))))$
   $= P(T_2, C_3, P(T_2, C_3, (T_6 - \{v\})) \cup P(T_2, C_3, \{f\}))$
   $= P(T_2, C_3, P(T_2, C_3, \{u\}) \cup \{j\})$
   $= P(T_2, C_3, \{h\} \cup \{j\}) = \{h, j\}$

Note that the not operation produces a complement relative to the nodes in type T_6.

E5. PRINT $A_{2,1}$ WHERE NOT ($A_{6,1}$ EQ GM OR $A_{1,1}$ EQ AUSTIN)

1. $T_A = T_2$
2. Types T_1, T_2, and T_6 uniquely determine context C_3 for this expression.
3. F(PRINT $A_{2,1}$ WHERE NOT($A_{6,1}$ EQ GM OR $A_{1,1}$ EQ AUSTIN)

$=P(T_2,C_3,(U-F(A_{6,1}$ EQ GM OR $A_{1,1}$ EQ AUSTIN)))

$=P(T_2,C_3,\{g,h,i,j\} -(P(T_2,C_3,F(A_{6,1}$ EQ GM))

$P(T_2,C_3,F(A_{1,1}$ EQ AUSTIN)))

$=P(T_2,C_3,\{g,h,i,j\} -(\{g\} \cup \{j\}))$

$=P(T_2,C_3,\{h,i\}) = \{h,i\}$

E6. PRINT $A_{2,1}$ WHERE (NOT $A_{6,1}$ EQ GM) AND (NOT $A_{1,1}$ EQ AUSTIN)

1. $T_A = T_2$
2. Types T_1, T_2, and T_6 uniquely determine context C_3.
3. F(PRINT $A_{2,1}$ WHERE (NOT $A_{6,1}$ EQ GM) AND (NOT $A_{1,1}$ EQ AUSTIN)

$=P(T_2,C_3,((U-P(T_2,C_3,\{v\})) \cap (U-P(T_2,C_3,\{f\}))))$

$=P(T_2,C_3,(U-\{g\}) \cap (U-\{j\}))$

$=P(T_2,C_3,\{h,i,j\} \cap \{g,h,i\}) = \{h,i\}$

Notice that, in contrast to example E4, the complementation here is with respect to U, which is the projection, as defined in semantic rule F9, of the types

in the qualifying clause, in context C_3.

E7. PRINT $A_{2,2}$ WHERE ($A_{4,1}$ EQ DALLAS AND ($A_{2,2}$ HAS $A_{5,1}$ EQ 416))

1. $T_A = T_2$

2. The types of the selection clauses, T_2 and T_4, and the type T_A uniquely determine the context C_2.

3. In the second selection clause, the types of the attributes $A_{2,2}$ and $A_{5,2}$ uniquely determine the context of that selection clause as C_4.

4. F(PRINT $A_{2,2}$ WHERE ($A_{4,1}$ EQ DALLAS AND ($A_{2,2}$ HAS $A_{5,1}$ EQ 416))

$=P(T_2,C_2,(P(T_2,C_2,\{q\}) \cap P(T_2,C_4,F(A_{5,1} \text{ EQ } 416))))$

$=P(T_2,C_2,(\{j\} \cap P(T_2,C_4,\{t,x\})))$

$=P(T_2,C_2,(\{j\} \cap \{g,j\})$

$= \{j\}$

E8. PRINT $A_{2,1}$ WHERE (($A_{2,2}$ HAS $A_{6,1}$ EQ ATT IN C3) OR ($A_{3,2}$ EQ 30K or ($A_{3,2}$ HAS $A_{2,1}$ EQ COLORADO))) OR ($A_{6,1}$ EQ GM)) IN C5

1. $T_A = T_2$

2. Context C_5 is determined by the IN C5 phrase. Recall that context C_5 is defined by $T_2T_3T_6$.

3. The projection for $A_{2,2}$ HAS $A_{6,1}$ EQ ATT IN C3 must occur in context C_3, or $T_1T_2T_6$.

4. Let the retrieval expression be denoted by Q.

$F(Q) = P(T_2,C_5,(P(T_2,C_5,F((A_{2,2}$ HAS $A_{6,1}$ EQ ATT IN C3) OR ($A_{3,2}$ EQ 30K or ($A_{3,2}$ HAS $A_{2,1}$ EQ COLORADO))))$\cup$ $P(T_2,C_5,F(A_{6,1}$ RQ GM)))

$=P(T_2,C_5,P(T_2,C_5,P(T_2,C_5,F(A_{2,2}$ HAS $A_{6,1}$ EQ ATT IN C3)) $\cup P(T_2,C_5,F(A_{3,2}$ EQ 30K)$\cup P(T_3,C_4,F(A_{2,1}$ EQ COLORADO))))

$\cup P(T_2,C_5,F(A_{6,1}$ EQ GM)))

$=P(T_2,C_5,P(T_2,C_5,P(T_2,C_5,P(T_2,C_3,F(A_{6,1}$ EQ ATT))).

$\cup P(T_2,C_5,\{1\} \cup P(T_3,C_4,\{h\}))) \cup P(T_2,C_5,\{v\}))$

$=P(T_2,C_5,P(T_2,C_5,P(T_2,C_5,P(T_2,C_3,\{u\}))$

$P(T_2,C_5,\{1\} \cup \emptyset)) \cup \{i,j\})$

$=P(T_2,C_5,P(T_2,C_5,\{h\} \cup \{g\}) \cup \{i,j\})$

$=P(T_2,C_5,\{g,h,i,j\}) = \{g,h,i,j\}$

Note that the projection of node u to type T_2 in context C_3 is h, and in context C_5 it is g, which is why the IN C_3 phrase is used.

In summary, one approach to the problem of Boolean processing on the acyclic data base is to augment and redefine the retrieval expression as found in Lowenthal (L4) to provide a basis for predictable results, and to make the processing of the expression workable

with respect to data structures based on the acyclic network. Bang's result (Appendix A) concerning the non-homomorphic nature of hierarchical dataset qualification leads to a separation of the Boolean operators into two sets, and, or, not, operating at the "selection" level, on datasets or nodes of a single type, and AND, OR, NOT, operating also within a single type, but upon sets of relatives of the selected nodes. Thus within the type specified in set A of the retrieval expression, there will be a set of nodes which satisfy the qualifying clause. This is essentially the "meaning" of a retrieval expression in a particular data base.

In the following chapter, another approach to the retrieval problem uses a set of procedures which are capable of doing Boolean as well as non-Boolean processing on networks.

CHAPTER IV

A PROCEDURAL APPROACH TO RETRIEVAL

In a recent dissertation, Hardgrave (H2) defines a set manipulation procedural language called BOLTS, for Boolean Oriented Language for Tree Structures. Although this language was originally designed to permit both Boolean and non-Boolean retrieval processing on tree-structured data bases, with slight modification it serves as a procedural language for retrieval processing in the acyclic data base. Such a language is useful in implementation because it provides a ready-made syntax for insertion into a programming language such as COBOL or FORTRAN; and because it allows many "non-standard" retrieval actions to be designed, such as searching through the data base, or simple "browsing" much as one would do in a library.

BOLTS is defined in terms of two sets of functions, set manipulation functions, and node extraction functions. The set functions are straightforward, and defined in terms of two sets of nodes, S_1 and S_2, they are

(1) UNION $(S_1, S_2) = S_1 \cup S_2$

(2) INTER $(S_1, S_2) = S_1 \cap S_2$

(3) DIFFER $(S_1,S_2) = S_1 - S_2$.

Node extraction functions TYPE, SELECT, and ADJUST are defined by Hardgrave to provide the following sets:

(1) TYPE(i) is the set of all nodes of type T_i,

(2) SELECT((a,r,v)), where a is an attribute identifier, r is a relational operator, and v is a value identifier, is the set of all nodes where SAT(arv) is true.

(3) ADJUST(S,i), where S is a set of nodes, and i is an integer, is the set of all nodes that are of type T_i and are hierarchically related as ancestors or descendants to the nodes in set S.

In terms of sets defined in this paper, TYPE(i) = typeset T_i, SELECT(a,r,v) = SEL(S,T) where S = <a><r><v> and a is an attribute of type T. The function ADJUST(S,i) is roughly correspondent to the projection function P defined in Chapter III. What ADJUST lacks is a context parameter to specify which kinds of paths are to be traversed in finding a projection. The projection function P(T,C,S) finds all of the nodes of type T which are related to nodes in the set S, traversing only those node sequences which conform to context C. Thus, by redefining ADJUST with an additional parameter, BOLTS may be expanded slightly to function in the acyclic data base. We define

$$ADJUST(S,i,C) = P(T_i,C,S),$$

where C is a context, and S is a set of nodes.

Since the projection mapping P has a somewhat complicated definition, involving the consideration of the node sequences it uses, the implication of this redefinition is that ADJUST for the acyclic data base will need to be assisted by some form of structural information in order to be efficient. We cannot design such a function and expect it to operate will if it must check each and every node sequence for contextual validity.

The crucial difference in Hardgrave's treatment of ADJUST and that given here lies in the concept of the broom, which Hardgrave defines as a node plus all of its ancestors and descendants. Since Hardgrave assumes that all hierarchical relationships are transitive, he is able to greatly reduce the amount of data necessary to represent various node sequences, or paths, in the tree, and since we make no such assumption in the network, it is conceivable that radically different methods of implementing the ADJUST function must be invented. There is nothing useful which is analagous to brooms in the acyclic data base. We have, at best, the possibility of imposing various orderings on contexts, c-sets, types, and nodes to arrange the structural information of the acyclic data base in an efficient way.

If we consider a revised BOLTS definition, using the six procedures

UNION(S_1,S_2),

INTER(S_1,S_2),

DIFFER(S_1,S_2),

TYPE(i)

SELECT(a,R,v), and

ADJUST(S,i,C),

then the retrieval function F defined in Chapter III may be recursively defined in terms of these six procedures, as follows:

B1. If S is a simple condition of the form <a><R><v>, then

F(S) = SELECT (a,R,v)

B2. If <a> HAS <qc> is a selection clause where attribute a is of type T_i and C is a context uniquely determined by T and the types of the selection clauses making up <qc>, then

F(<a> HAS <qc>) = ADJUST(F(<qc>),i,C)

B3. If (<a> HAS<qc>IN<C>) is a selection clause where a is of type T_i, and the clause is valid by rule S10, then

F(<a>HAS<qc>IN<C>) = ADJUST(F(<qc>),i,C)

B4. If S is a selection clause, then

F((X)) = F(S)

B5. If S_1 and S_2 are selection clauses of type T, then

$F(S_1 \text{ or } S_2)$ = UNION $(F(S_1),F(S_2))$

$F(S_1 \text{ and } S_2)$ = INTER $(F(S_1),F(S_2))$

B6. If S is a selection clause of type T_i, then

F(not S) = TYPE(i) - S

For the following definitions, if Q is a qualifying clause preceded by HAS, then T_z denotes the type of the attribute preceding the HAS operator. If Q is not preceded by HAS, then T_z denotes the type of the attribute set A, or T_A.

B7. If Q is a qualifying clause, then

F((Q)) = F(Q)

B8. If Q_1 and Q_2 are qualifying clauses of types T_i and

T_j, respectively, and the context C is uniquely determined by T' and the types of the selection clauses making up Q_1 and Q_2, then if Q is of the form Q_1 OR Q_2, then

$$F(Q) = F(Q_1 \text{ OR } Q_2) = \text{UNION (ADJUST}(F(Q_1),z,C),$$
$$\text{ADJUST}(F(Q_2),z,C)), \text{ and}$$

if Q is of the form Q_1 AND Q_2, then

$$F(Q) = \text{INTER(ADJUST}(F(Q_1),z,C),\text{ADJUST}(F(Q_2),z,C))$$

B9. If Q is a qualifying clause composed of selection clauses of types $T_{i_1},\ldots,T_{i_k}$, and C is the context which is uniquely determined by T_z and $T_{i_1},\ldots,T_{i_k}$, then NOT implies complementation in the universe U defined as

$$U = \bigcup_{j=1}^{k} \text{ADJUST}(T_{i_j},z,C), \text{ and}$$
$$F(\text{NOT } Q) = U - \text{ADJUST}(F(Q),z,C)$$

B10. If RE is a retrieval expression, the attribute set A is of type T_z, and the context C is uniquely determined either by the type T_z and the types of the selection sets making up qc , or by an IN C phrase, then

$$F(RE) = \text{ADJUST}(F(\langle qc \rangle),z,C)$$

Thus, BOLTS may be used in its modified form to simulate the non-procedural language defined in Chapter III. It is important to note that BOLTS is capable of doing much more than just this, however. It is fully general procedural system which can be used to manipulate sets of nodes in any way the user desires.

Because of the complexity of finding ancestors and descendants in the acyclic data base, and because such an operation is necessary both in the procedural and non-procedural languages defined, it is necessary to consider a structural representation which will tend to aid in the functioning of such languages. Chapter V is a discussion of a structural representation with this capability.

CHAPTER V

STRUCTURAL REPRESENTATION

The structural characteristics of the finite acyclic network with which this paper deals can readily be explicitly represented by a system which uses pointers, with the arrows in the various diagrams being represented by the address of the node at the "head" of each arrow. Such mechanisms have been implemented for tree-structured data management systems, for example, in System 2000(S6), and in a system described in a paper by Landamer(L1), and in at least one network-like system, the Integrated Data Store (B1). These "pointer" systems typically provide reasonable retrieval times and fairly cheap updating procedures. However, in large data bases, rather complicated schemes are necessary to preserve the pointer structure, with a consequent increase in retrieval times. We therefore seek an implicit structural representation for the acyclic data base which is not pointer-oriented in the belief that this type of representation will provide an opportunity for consistently better retrieval times as the acyclic data base gets larger and older. It is hoped that these fundamental definitions will provide a basis for further study and possibly an effective

implementation of an efficient network-structured data management system.

The Slot Concept

Since each typeset consists of a finite number of nodes, it is natural to set up a finite number of available slots for each type, with each slot being represented by an integer. The treatment of the slot as an abstract entity rather than as an implementation "worry" has several advantages:

(1) The amount of storage needed for structural representation may be estimated accurately only when the cardinality (number of slots) for each type is pre-defined.

(2) For each type, each slot may be easily represented by an integer, which allows an internal ordering of slots which amounts to an ordering of nodes which is transparent to the data base user. This means that in the view of the user, the nodes are essentially unordered within a typeset.

(3) The assumption of a finite number of "slots" for each type admits considerations of various techniques which have so far been applied only slightly to data management, i.e.,

(i) sparse storage techniques (such as in sparse matrices)

(ii) virtual storage management techniques, akin to virtual memory techniques.

Path Representation

Since each node within a typeset is unique, and each slot number is unique, it is natural to form a 1:1 correspondence between nodes within a typeset and a set of slot numbers for that type. Therefore, each node label as depicted in Figures 3,5,7, and 8, may be identified by the specification of a type label and a slot number.

For a given sequence of m types, ordered from the type of lowest level to the type of highest level, $T_{j_1},\ldots,T_{j_m}$, a path is a sequence of m integers $i_1,\ldots,i_m$, such that i_k is the slot number of a node of type T_{j_k}, for k=1,...,m. With this basic definition in mind, the function of a data base loader is to specify the contents of individual nodes and the paths between individual nodes. The function of a structural representation is to store the information supplied by the data base loader, in such a way that the representation may act as an aid to the retrieval operation.

Lowenthal (L4) points out the information-richness of a similar construct, the trace, which allows efficient retrieval operations. The attractiveness of this scheme prompts the development of a similar construct

TYPE	NODE	POSITION(SLOT NO.)
	x	6
	y	9
T_6	u	6
	v	8

The table above and on the previous page indicates that node a occupies slot 2, type T_1, node b occupies slot 3, type T_1, etc. The nodes may be ordered within a type by their slot number.

The node sequence set NS, in Figure 8 lists the sequence (i,o,w). These nodes are of types T_2, T_3, T_5, respectively, so that the sequence exists in context C_4. The sequence of slot numbers corresponding to this node sequence is (6,3,5). Here m = 3 from the length of the context definition, and by setting d_k to 10^{m-k}, we have

$$t = \sum_{k=1}^{3} d_k i_k = 10^2 \times 6 + 10^1 \times 3 + 10^0 \times 5 = 635.$$

Similarly, the node sequences (e,w) and (f,w) are elements of NS also, and from the position table, these produce traces for w of 75 and 95, respectively, in context C_1. The c-trace for node w would therefore be 75, 95, in context C_1.

Structure table

The previous listing suggests a structure table to record the association of contexts, types, nodes, and traces, with appropriate embellishment to allow for groupings of traces and contexts, according to the following syntactic rules:

<c-trace> ::= <trace> | <trace> , <c-trace>

<l-pair> ::= <node label> < c-trace >

<pair list> ::= <l-pair> | <l-pair> <pair-list>

<type group> ::= <type label> <pair list >

<t-list> ::= <type group> | <type group> <t-list>

<c-group> ::= <context label> < t-list >

<structure table> ::= <c-group> | <c-group> <structure table>

The above syntax includes the node label in the structure table construction, which is an obvious redundancy, since for each slot number in a type there may only be one node label, and vice versa. However, the inclusion of the node label makes the structure table easier to follow, and thus it has been included. No doubt there will be cause for eliminating this label in some future implementation.

The information contained in the structure table is intended to record the paths, if any, to each node from

those in higher (lower numeric) levels, and to indicate when the node is not connected hierarchically to any node of a higher level. This may be formally defined in three stages, as follows:

Suppose x is a node of type T_p, C is a context with definition $T_{j_1} \ldots T_{j_m}$, where $j_1 \leq p \leq j_m$, and SSN = $\{(i_{1,1},\ldots,i_{1,m}),\ldots,(i_{n,1},\ldots,i_{n,m})\}$ denotes the set of n slot number sequences, each m integers in length, which are to be associated with node x. Then

(1) if x does not appear in the c-set associated with context C, then n = 0 and no slot number sequences are defined, otherwise

(2) if x appears in the c-set associated with C but not in a node sequence in the c-set, n = 1, and $i_{1,k}$ = 0 except when k = p, for k = 1,...,m, and when k = p, $i_{1,p}$ = the slot number of x, otherwise

(3) n is the number of node sequences in the c-set associated with C in which x appears, and for r = 1,...,n

(i) $i_{r,k}$ = 0, for $k > p$,

(ii) $i_{r,p}$ = slot number of x,

(iii) for q = 1,...,p - 1,

$$i_{r,p-q} = \begin{cases} 0, \text{ if } i_{r,p-q+1} = 0, \text{ or if there is no node q places to the left of x in the node sequence, otherwise} \\ \text{the slot number of the node q places to the left of x in the node sequence} \end{cases}$$

Each member of the set SSN is a slot number sequence which corresponds to a node sequence containing x, for a particular context. The traces to be associated with x in the structure table are the integers generated by the sequence of sums

$$t_r = \sum_{j=1}^{m} d_j i_{r,j}, \quad r = 1,\ldots,n, (d_j \text{ pre-defined})$$

If n = 0, we do not store a trace for x relative to context C. There will be at least one non-empty trace for x, therefore. Also, if two slot number sequences produce the same trace, it is stored only once in the table.

The following is a structure table so derived from the nodes and slot numbers listed above, based on the c-sets of Figure 8:

Structure Table from Figure 8

Context	Type	Node	c-trace	Corresponding node sequences
C_1	T_1	a	20	
		b	30	
		c	40	
		d	50	
		e	70	
		f	90	
	T_5	s	31	(b,s)
		t	32	(a,t)
		w	75,95	(e,w),(f,w)
		x	06	
C_2	T_2	g	20	
		h	30	
		i	60	
		j	80	
	T_4	k	23	(g,k)
		m	24	(g,m)
		q	87	(j,q)
C_3	T_1	a	200	
		b	300	
		c	400	
		d	500	
		e	700	
		f	900	
	T_2	g	220,320	(a,g),(b,g,v)

Context	Type	Node	c-trace	Corresponding node sequences
		h	430	(c,h,u)
		i	560,760	(d,i),(e,i)
		j	980	(f,j)
	T_6	u	436	(c,h,u)
		v	328	(b,g,v)
C_4	T_2	g	200	
		h	300	
		i	600	
		j	800	
	T_3	l	210	(g,l,s),(g,l,t)
		n	620	(i,n)
		o	630	(i,o,w)
		p	840	(j,p,x),(j,p,y)
		r	850	(j,r)
	T_5	s	211	(g,l,s)
		t	212	(g,l,t)
		w	635	(i,o,w)
		x	846	(j,p,x)
		y	849	(j,p,y)
C_5	T_2	g	200	
		h	300	
		i	600	
		j	800	
	T_3	l	210	(g,l,u)

Context	Type	Node	c-trace	Corresponding node sequences
		n	620	(i,n,v)
		o	030	
		p	040	
		r	850	(j,r,v)
	T_6	u	216	(g,l,u)
		v	628,858	(i,n,v),(j,r,v)

The data structure in Figure 8 is relatively simple. Application of trace and context concepts to a more complex example may be useful. Consider the following hypothetical situation:

a manufacturer defines a hierarchical arrangement of entities he terms as suppliers, parts, subassemblies, components and buyers. These are represented as types.

T_1 - suppliers
T_2 - parts
T_3 - subassemblies
T_4 - components
T_5 - buyers

The term repeating group, now commonly used in data management theory, will be used in the following sense. To say that T_1 and T_2 are types and T_2 is a re-

peating group with respect to T_1 means that for each node of type T_1 there may be many nodes of T_2 associated with it. Notice that in the acyclic data base there is no limitation upon the levels of T_1 and T_2.

The manufacturer requires a data base which will reflect the following relationships:

(1) several suppliers sell several parts each, implying parts must be a repeating group with respect to suppliers,

(2) several parts come from various suppliers, implying suppliers must be a repeating group with respect to parts,

(3) a part may be used in several subassemblies, implying subassemblies must be a repeating group with respect to parts,

(4) most subassemblies require several parts, implying parts must be a repeating group with respect to subassemblies,

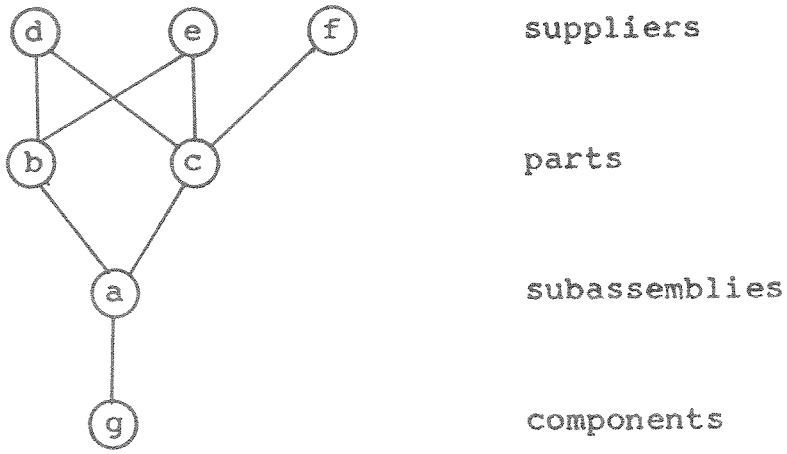
(5) some subassemblies are used in several components, implying components must be a repeating group with respect to subassemblies,

(6) most components require several subassemblies, implying subassemblies must be a repeating group with respect to components,

(7) there are gaps in the structure, i.e., some

parts may not be used in any subassembly, or some component may not require any subassembly, or there may be suppliers listed who do not currently supply anything,

(8) there is a need for <u>specificity</u> about which parts are used in which components, for example, the manufacturer finds it more economical to build subassembly a with parts b or c from suppliers d, e, or f, but has no market for component g unless it is specifically constructed with subassembly a with part b <u>only</u>, supplied by supplier f. Thus, for example, the existence of the structure



in the network must definitely be controlled to eliminate all of the extraneous paths to node g,

(9) each component may be sold to several buyers, and each buyer may require several components, so that T_4 and T_5 must be repeating groups with respect to each other.

We are thus led to the following network structure:

T_1: SUPPLIERS (RG in PARTS)

T_2: PARTS (RG in SUPPLIERS, RG in SUBASSEMBLIES)

T_3: SUBASSEMBLIES (RG in PARTS, RG in COMPONENTS)

T_4: COMPONENTS (RG in SUBASSEMBLIES, RG in BUYERS)

T_5: BUYERS (RG in COMPONENTS)

There is a single context: $T_1T_2T_3T_4T_5$.

Figure 10 will serve as a concrete example of just such a structure, with the various exclusions listed, rather than labelling the graph.

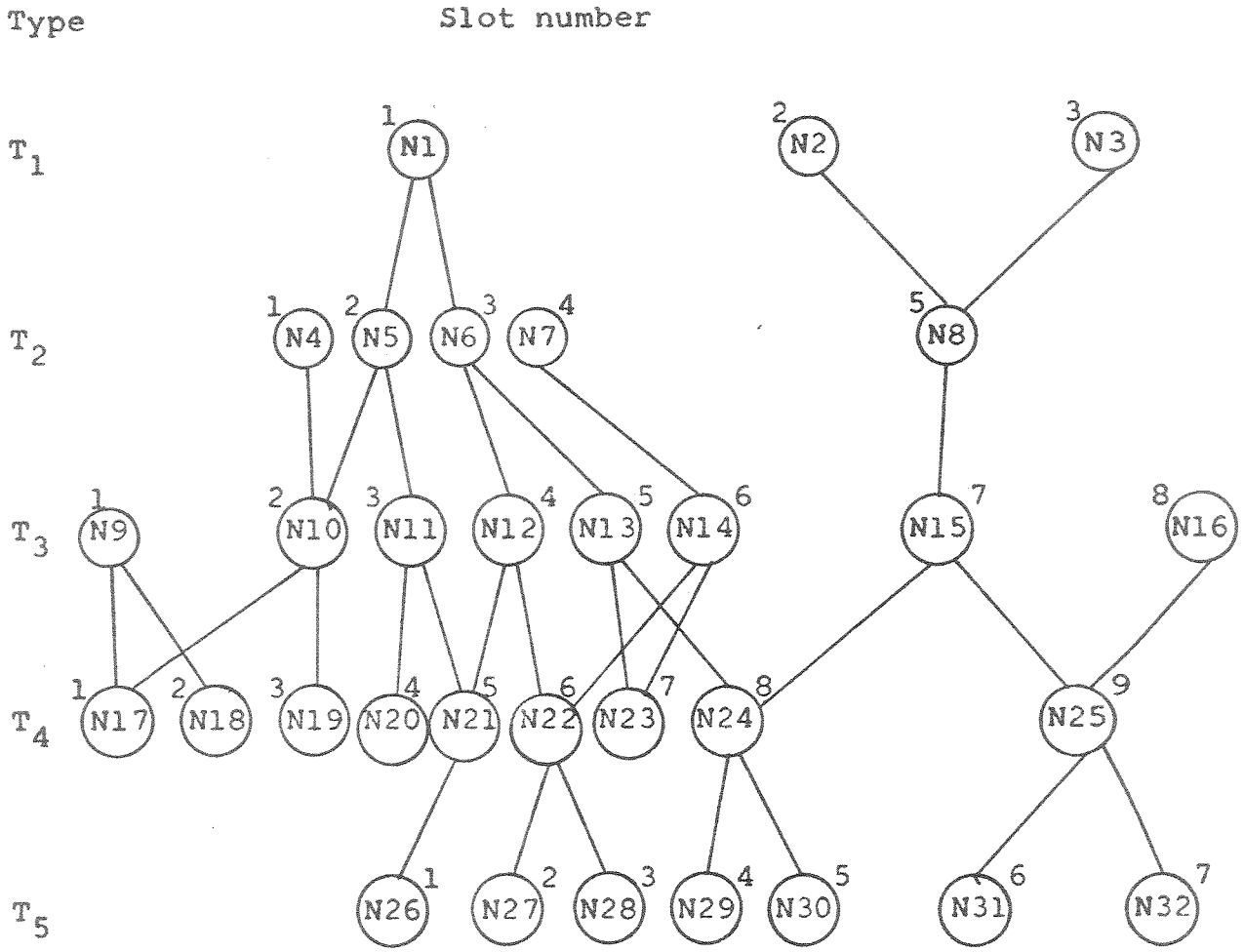
In developing the structure table for the Figure 10 context, it is assumed for the sake of simplicity that each nodes' slot number in its type is the position of the node in the type from the left, as indicated in the figure. Figure 11 shows the structure table for the context. Note the absence of traces 01210, for node N17, 04662, for node N27, 35780, for node N24, and as a result, the absence of traces 35784 and 35785 for nodes N29 and N30, respectively. These absences are a direct result of the path exclusions mentioned in Figure 10.

The specification of which paths between data-sets are to be excluded must occur at the time the data is actually loaded. This, of course implies that there must exist some data input language capable of excluding

certain paths or specifying exactly those paths which are pertinent.

In summary, the trace device developed is an appropriate mechanism to store path information in the acyclic network. The a priori specification of the number of slots for nodes of a single type is desirable from the designer's point of view because it means his specification for the storage structure representing the trace table may be based on this information. There is a correspondence between the retrieval ecpression thus far defined and the structure table, because the retrieval expression is designed to be processed by c-sets, and the structure table is organized on the basis of c-sets.

In the next chapter, the actual basic algorithms of retrieval processing with the structure table and an inverted file are presented.



PATH EXCLUSIONS:

1. Buyer N27 will buy component N22 only if it is made with subassembly N12.
2. Component N17 made with subassembly N10 uses part N5 only, not part N4.
3. N3 supplies N8 for N25 only, not N24.

NOTE: This is a graph of a single context.

Sample Context

Figure 10

Type	Node	c-trace
T_1	N1	10000
	N2	20000
	N3	30000
T_2	N4	01000
	N5	12000
	N6	13000
	N7	04000
	N8	25000,35000
T_3	N9	00100
	N10	01200,12200
	N11	12300
	N12	13400
	N13	13500
	N14	04600
	N15	25700,35700
	N16	00800
T_4	N17	00110,12210
	N18	00120
	N19	01230,12230
	N20	12340
	N21	12350,13450
	N22	13460,04660

Figure 11

Type	Node	c-trace
	N23	13570,04670
	N24	13580,25780
	N25	25790,35790,00890
T_5	N26	12351,13451
	N27	13462
	N28	13463,04663
	N29	13584,25784
	N30	13585,25785
	N31	25796,35796,00896
	N32	25797,35797,00897

Figure 11
(continued)

CHAPTER VI

RETRIEVAL ALGORITHMS

Introduction

We assume the existence of three files in this chapter, the structure file, the inverted file, and the data file. These files are usually implemented on mass storage devices for data management systems, and serve the following essential functions:

(1) The structure file is the storage place for relationships between datasets. It does not contain the actual data, but rather some entity which corresponds to a dataset, a label, a pointer, an address, etc. The structure file is assumed to be as defined in Chapter V, which utilizes traces grouped by c-sets, and augmented by a pointer to the location of the dataset in the data file comprising the content of the node represented by the set of traces in the table.

(2) The inverted file is a representation of a mapping from a/v pairs to nodes or to a set which corresponds to the node set. It is desirable to have the initial retrieval operators SEL or SELECT access the inverted file to give the informations about those datasets which satisfy simple conditions.

(3) The data file is where the actual data values (or references to data literals) are stored, grouped by datasets. The ordering of datasets in the file may be arbitrary.

The inverted file and selection operations

At the root of retrieval processing is the selection of those nodes satisfying a simple condition aRv, where a is an attribute, v, a value, and R is a relational symbol. We assume the existence of an inverted file which serves as a helpful tool in finding those nodes directly. For a given simple condition aRv, the inverted file must contain structural information about those nodes which (1) contain attribute a, and (2) contain a value v' paired with a such that v' and v may be compared to see if v' bears the relation R to v. The structural information for the nodes is necessary to form Q-projections into a given type, in the processing generated by the qualifying clause.

The function SEL, defined in Chapter III as

$$SEL(S,T) = \{n \in T \mid SAT(aRv,n) = 1\}$$

is the set-theoretic node selector for a simple condition aRv. However, for the retrieval algorithms, the structural information contained in the inverted file

a v_1 c-trace for g_1,...,c-trace for g_k

v_2 c-trace for g_1,...,c-trace for g_k

⋮

v_j c-trace for g_1,...,c-trace for g_k

If a type T has m attributes, then these are grouped together as follows:

T entry for a_1

entry for a_2

⋮

entry for a_m

For convenience, in Figure 13, the context groups are numbered, to show a typical inverted file representation, based on the example in Figure 8. It is usually convenient to sort the values associated with each attribute, to allow a binary search.

Algorithm R1 produces a set of traces from the inverted file, when given a context label C and a simple condition S = aRv.

Algorithm R1

R1.1. Select the context group G associated with the type T of a which includes context symbol C.

R1.2. Search for type T in the inverted file. Select the traces in context group G associated with attribute a and value v' such that v' bears the relationship R to v.

R1.3. This set of traces is denoted as R1(aRv).

Examples of R1

1. $A_{5,1}$ EQ 211, in context C_5. This simple condition is of type T_5. The context group in the table in Figure 13 associated with type T_5 and which includes context C_5 is G4. According to the definition of an inverted file entry, the traces associated with $A_{5,1}$ in the context group G5 are listed in the second group. Thus, R1($A_{5,1}$ EQ 211) = (635,849).

2. $A_{2,1}$ EQ ILLINOIS, in context C_3. The type T_2 has three context groups associated with it, but only G3 contains the context C_3. In the inverted file, associated with the a/v pair $A_{2,1}$/ILLINOIS, in the second position, are the traces (220,320,420).

Algorithm R1 satisfies the need in the retrieval mapping defined in Chapter III for a primitive selection condition SEL. Rules F5, F6, F8, and F9 for the retrieval mapping F demand algorithms which perform the standard set operations of union, intersection and complement. The rules F2 and F3, for the HAS operator, require an algorithm to form the projection of a set

Context Groups	Type Inclusions
G1 = (C_1)	T_1 is in G1,G3
G2 = (C_2)	T_2 is in G2,G3,G6
G3 = (C_3)	T_3 is in G6
G4 = (C_4)	T_4 is in G2
G5 = (C_5)	T_5 is in G1,G4
G6 = (C_4,C_5)	T_6 is in G3,G5

Inverted File from Fig. 8 Data Base

Type	Attribute	Value	c-traces
T_1	$A_{1,1}$	AUSTIN	(90),(900)
		CHICAGO	(30),(300)
		CODY	(50),(500)
		COLO. SPRINGS	(40),(400)
		JACKSON HOLE	(70),(700)
		SPRINGFIELD	(20),(200)
	$A_{1,2}$	1024 WALTERS BLVD.	(20),(200)
		203 STATE ST.	(30),(300)
		36 N. MAIN	(50),(500)
		406 3RD ST.	(40),(400)
		804 CONGRESS	(90),(900)
		<null>	(70),(700)

Figure 12

	$A_{1,3}$	446-7213	(90),(900)
		462	(50),(500)
		672-7143	(30),(300)
		733-6111	(40),(400)
		756-0934	(20),(200)
		<null>	(70),(700)
T_2	$A_{2,1}$	COLORADO	(30),(430),(300)
		ILLINOIS	(20),(220,320),(200)
		TEXAS	(80),(980),(800)
		WYOMING	(60),(560,760),(600)
	$A_{2,2}$	11M	(20,80),(220,320, 980),(200,800)
		2.6M	(30),(430),(300)
		314K	(60),(560,760), (600)
T_3	$A_{3,1}$	1	(630)
		2	(620)
		420	(840)
		462	(210)
		601	(850)
	$A_{3,2}$	14K	(630)
		15K	(840)
		30K	(210)
		<null>	(850)
	$A_{3,3}$	JONES,X.Y.	(210)
		SMITH,A.	(620)

Figure 12
(continued)

		SNORT,S.S.	(630)
		THOMPSON	(840,850)
T_4	$A_{4,1}$	CHICAGO	(23)
		DALLAS	(87)
		PEORIA	(24)
	$A_{4,2}$	136K	(24)
		3.46M	(23)
		795K	(87)
	$A_{4,3}$	2.3	(24)
		3.4	(87)
		<null>	(23)
T_5	$A_{5,1}$	211	(75,95),(635,849)
		415	(31),(211)
		416	(32,06),(212,846)
	$A_{5,2}$	AGAINST	(22,32,75,95), (212,635)
		FOR	(31,06),(211,846)
		<null>	(09),(849)
T_6	$A_{6,1}$	ATT	(436),(216)
		GM	(228,328),(628)

Figure 12
(continued)

of traces, and possibly to change the traces after projection to those of another context. This would be the case, for example, in example E7, p. , where the context changes from $C_4(A_{2,2}$ HAS $A_{5,1}$ EQ AUSTIN), to C_2(PRINT $A_{2,2}$ WHERE ($A_{4,1}$ EQ...)). Qualifying clauses based on the operators AND, OR, NOT require projections to type T', whether T' is T_A or the type of an attribute preceding a HAS.

Therefore, there are three basic algorithmic mechanisms required by the retrieval mapping. They are

(1) a procedure to form unions, intersections, and complements of sets of traces,

(2) a procedure to form the projection from a set of traces of one type to traces of another type, and

(3) a procedure to find the traces in one context when given traces of the same node in another context.

Relationships of traces

Within each context, a trace may be associated with only one type in the structure table. Since set operations on traces produced from the inverted file by algorithm R1 are to be defined, it is useful to attach a context-type identity to trace sets selected by R1. We therefore define the c-type of a set of traces selected

from the inverted file from a simple condition of type T_j, in context C_i, to be the ordered pair (i,j).

For example, the trace set (635,849) produced from the simple condition $A_{5,1}$ EQ 211, in context C_5, is of c-type (5,5). Likewise, the trace set (220,320, 420) produced from the simple condition $A_{2,1}$ EQ ILLINOIS, in context C_3 is of c-type (3,2).

The definitions of relationships between traces which are needed for the three algorithmic mechanisms described above are (1) equality, (2) ancestry, and (3) context shift. A definition of equality is needed to define set operations on trace sets. A definition of ancestry is needed to form projections of traces. A definition of context shift is needed, to allow algorithmic definition of projections implied by the HAS operator, since it is entirely possible that the context of a selection clause <a> HAS <qc> and the context of <qc> may be different (see example E7, Chapter III). We therefore make the following definitions:

(1) Equality.

If t_1 and t_2 are traces of c-types (i,j), (k,m), respectively, then $t_1 = t_2$ only in case i = k, j = m, and the respective fields of t_1 and t_2 are equal.

(2) Ancestry.

If trace t_1, formed by the concatenation

of fields $I_1,\ldots,I_k$, is of c-type (i,j), and trace t_2, formed by the concatenation of fields $J_1,\ldots,J_k$, is of c-type (k,m), are such that there exists an integer p such that $I_n = J_n$, n = 1,...,p, and $I_n = 0$ for all p n k, and i = k, then t_1 is said to be an ancestor of t_2 and the predicate $ANC(t_1,t_2)$ has a true value. The statement that t_1 is a descendant of t_2 may be expressed as $ANC(t_2,t_1)$.

(3) Context shift.

Suppose that context C_p is defined as $T_{i_1}\ldots T_{i_n}\ldots T_{i_k}$, and C_q is defined as $T_{j_1}\ldots T_{j_m}\ldots T_{j_k}$ where $i_n = j_m$, and t is a trace of c-type (p,i_n). Then all traces of c-type (q,j_m) in the structure table are in the context shift of trace t from C_p to C_q.

Examples

Referring to the structure table illustrated on p. , trace 849, of c-type (4,5), is a descendant of trace 840, also of c-type (4,5) because the fields compare identically from the left, or are zero in the higher trace, 840.

Consider the context $C_4{:}T_2T_3T_5$ and the context $C_3{:}T_1T_2T_6$ for the same structure table. We wish to find the context shift from C_4 to C_3 of the trace 600 of c-type (4,2). Traces of c-type (3,2) which have a 6 in the second field are 560, 760, and these form the context shift.

Algorithm UI

Algorithm UI is used to form the union and intersection of a pair of trace sets of the same type, and in the same context. We assume that Q = $q_1,\ldots,q_m$ and R = $r_1,\ldots,r_n$ are two sets of traces of the same type, and in the same context. We assume they are both sorted in ascending order. Algorithm UI follows.

UI.1. Set i to 1 and j to 1. Set U to $\emptyset$ and I to $\emptyset$.

UI.2. Set t to q_i. Set s to r_j.

UI.3. Add t to the set U.

UI.4. If t $\neq$ s, go to step UI.19.

UI.5. If t = s, go to step UI.12.

UI.6. If j = n, go to step UI.10.

UI.7. Set j to j+1.

UI.8. Set s to r_j.

UI.9. To step UI.4.

UI.10. Add the rest of set Q to U, in order.

UI.11. Stop.

UI.12. Add t to set I (intersection).

UI.13. If i=m go to step UI.24.

UI.14. If j=n go to step UI.10.

UI.15. Set i to i+1.

UI.16. Set j to j+1.

UI.17. Set t to q_i, and s to r_j.

UI.18. Go to step UI.3.

UI.19. Add s to the set U.

UI.20. If i=m go to step UI.

UI.21. Set i to i+1.

UI.22. Set t to q_i.

UI.23. Go to step UI.3.

UI.24. Add the rest of set R to U.

UI.25. Stop.

Complement within a type

Suppose $M = (t_1,\ldots,t_k)$ is a set of traces of c-type (i,j), sorted in ascending order, and N is the set of all traces of c-type (i,j). Let N be sorted in ascending order and represented by $N = (s_1,\ldots,s_j)$. Since $M \subseteq N$, $j \geq k$. The following algorithm COM, produces in the set C the complement of M with respect to type j in context C_i.

COM.1. If M is empty, go to COM.17.

COM.2. If k=j, go to COM.16.

COM.3. Set C to $\emptyset$.

COM.4. Set i and h to 1.

COM.5. Set m to t_h.

COM.6. Set n to s_i.

COM.7. If m=n go to COM.11.

COM.8. Add trace n to set C.

COM.9. Set i to i+1.

COM.10. Go to COM.6.

COM.11. If h k go to COM.15.

COM.12. Set i to i+1.

COM.13. Set h to h+1.

COM.14. Go to COM.5.

COM.15. Add the remaining traces in N to C. Stop.

COM.16. C is the empty set. Stop.

COM.17. C is the set N. Stop.

The Structure File and projection operations

The projection mapping P(T,C,S) defined in Chapter III requires an algorithm to form the projection of a set of traces of a single c-type. Suppose C_i is a context and $S = t_1,\ldots,t_k$ is an ordered set of traces of c-type (i,j). Let the number of type symbols in context C_i be m, so that each trace may be represented by m fields $f_1,\ldots,f_m$. The following algorithm produces $P(T_k,C_i,S)$:

P.1. If $LEV(T_k) < LEV(T_j)$ go to P.5.

P.2. If $LEV(T_j) < LEV(T_k)$ go to P.7.

P.3. The set S is the projection $P(T_k,C_i,S)$, because $LEV(T_k) = LEV(T_j)$.

P.4. Sort the traces in the structure table at type T_k in ascending order. Assign this set to T.

P.5. Using algorithm UI, find the intersection of the sets S and T, at step UI.5 comparing only the non-zero fields of the traces in the set T with the corresponding fields in traces in set S. This intersection is $P(T_k,C_i,S)$.

P.6. Stop.

P.7. Sort the traces in the structure table at type T_k in ascending order. Assign this set to T.

P.8. Using slgorithm UI, find the intersection of the sets S and T, at step UI.5 comparing only the non-zero fields of the traces in the set S with the corresponding fields of the traces in the set T. This intersection is $P(T_k,C_i,S)$.

P.9. Stop.

Example

For example, in Figure 11 consider the set S of traces of type T_5 (13463,25785,13585). We wish to project these to the traces at type T_3. Since $LEV(T_5)$ is greater than $LEV(T_3)$, the algorithm directs us to step P.7. The set S in sorted order is (13463,13585, 25785), and at P.7. this must be intersected with the non-zero fields of the traces at T_3, in sorted order, or (00100,00800,01200,04600,12200,12300,13400,13500,25700, 35700). The non-zero fields match for 13400, 13500, 25700, and therefore this is the projection of S into the traces

of type T_3.

Context Shift algorithm

Suppose that context C_p is defined as $T_{i_1} \cdots T_{i_n} \cdots T_{i_k}$, and context C_q is defined as $T_{j_1} \cdots T_{j_m} \cdots T_{j_k}$, where $i_n = j_m$, t is a trace of c-type (p,i_n). The following algorithm produces the context shift of t from context C_p to context C_q.

CS.1. Select the n^{th} field of trace t. Assign this to F.

CS.2. Among the traces of c-type (q,j_m) in the structure table, compare the m^{th} field with F. If it matches F, add the trace to the context shift of t. The set of traces so generated is the context shift of t from context C_p to context C_q.

Example

Consider the traces in the structure table for Figure 8, listed in Chapter IV. Let p=3, q=4, and let t=430. The definition for C_3 is $T_1T_2T_6$, and for C_4 is $T_2T_3T_5$. Therefore, by algorithm CS, we select the 2nd field of t, since $i_2=2=j_1$, and compare this field, or 3, with traces of c-type (4,2) in the structure table. These are 200, 300, 600, 800. The only trace which compares is 300. Thus 300 is the context shift of 430 from context C_3 to context C_4.

It is significant to note that these three algorithms are very simple. In a recent dissertation, Everett (E1) shows that additional structure in a data management system is likely to reduce retrieval effort. This is reflected in the simplicity of the algorithms just presented.

Examples

A few final examples will illustrate how the inverted file, structure table, and the algorithms on the previous pages might be used to process some retrieval expressions for Figure 8.

Example 1.

PRINT $A_{1,2}$ WHERE $A_{6,1}$ EQ GM

(1) The context is well-defined as C_3.

(2) The inverted file yields traces 228, 328 for $A_{6,1}$ EQ GM in context C_3.

(3) The attribute $A_{1,2}$ specified a projection to type T_1. From the structure table, the traces for type T_1 at context C_3 are 200, 300, 400, 500, 700, 900.

(4) Comparison on the non-zero fields of the two trace lists yields the intersection, which is 200, 300.

(5) Inverting these traces, we get nodes a,b.

Example 2.

PRINT $A_{2,1}$ WHERE ($A_{6,1}$ EQ ATT AND ($A_{2,1}$ HAS $A_{3,2}$ EQ 14K)) IN C_3

(1) The type $T_A = T_2$

(2) Context C_3 is specified, by the IN C_3 phrase. Context C_3 is defined as $T_1T_2T_6$. Note that the IN C_3 phrase is important here because there are relationships of nodes in type T_2 with nodes of type T_6 with nodes of type T_3 in between, but in context C_5.

(3) $R1(A_{6,1}$ EQ ATT) = (436), in context group G3.

(4) $R1(A_{3,2}$ EQ 14K) = (630), in context group G6. Note that here, the context of the selection clause $A_{2,1}$ HAS ($A_{3,2}$ EQ 14K) is ambiguous, but it does not matter, since the ambiguity involves types not used in the expression, T_5 and T_6.

(5) We project 630 to type T_2, using algorithm P and the structure table. In context C_4, $P(T_2,C_4,(630)) = 600$.

(6) The trace 600 of c-type (4,2) must be shifted in context to a trace set of c-type (3,2). Using algorithm CS, the context shift of

600 from C_4 to C_3 is (560,760).

(7) To carry out the intersection implied by AND, the trace (436) must be projected to T_A or T_2 by algorithm P, giving 430.

(8) The intersection of the two trace sets, (430) and (560,760), is empty. The retrieval expression therefore does not specify any dataset in the data base.

CHAPTER VII

CONCLUSION

It is concluded that the basic ideas for a graphical data structure which is (1) more general than a tree, (2) less general than a directed graph, and (3) imbued with enough potential internal ordering possibilities to allow reduction of retrieval times, have been explored to the extent that appropriate storage structures and algorithms suitable for data management appications have been identified. In fact, it has been demonstrated that the traditional tools of the data management system designer can be applied to this more general data structure.

Some important ideas have come out of this study. They are listed below.

(1) Because there are too many inherent possibilities for the grouping of type hierarchies in a data structure based on the acyclic digraph, the central idea of a general context is introduced to control these groupings.

(2) In the same spirit as the context idea is the use of a structure table based on traces of nodes

which exclude spurious hierarchical relationships, which would cause no problem in tree-like structures, but must be eliminated to get firm answers from the acyclic data base.

(3) It is demonstrated that there need be only simple algorithmic mechanisms to produce a data retrieval from a request. Although it is true that more structure is required than for tree-based systems, there are no doubt structures which exist which are seldom updated, or updated in a batch mode, or sparse enough to avoid a great deal of mass storage paging, and which also demand a richer structure than that available in a tree. For these structures, the acyclic network has a real chance of becoming cost effective. In this regard, a simple algorithmic process for retrieval keeps the operating cost down at the design stage, which is a useful characteristic to have when developing any prototype system.

(4) A redefinition of the syntax and semantics of the retrieval expression makes it applicable to the acyclic data base. It is demonstrated that this expression may be stated in terms of Boolean expressions of simple conditions, or in terms of a primitive set of procedures, the set described in the BOLTS language of Hardgrave (H2), with very slight modification.

In the area of this research, the following are some related tasks which could be explored:

(1) A model implementation needs to be made, to study the various implementational difficulties which inevitably occur, such as how to page the large files, the structure of the inverted file, the size of the trace fields, etc. It is felt that until some model can be constructed, further abstract treatment of the area of networks will be progressively more difficult to delineate. Also, a model would allow some experimentation with various forms of the retrieval expression, and the sub-problems of how to specify context, type projection, etc., could be approached on firmer ground.

(2) A study of the possibility of more parallel retrieval operations could lead to a higher efficiency in retrieval and possibly in updating. Given the hardware, the retrieval expression as defined in terms of selection sets, projections, and Boolean combinations of projected sets, could be speeded up considerably with the use of parallel operations to select and project traces.

(3) The problem of updating and loading trace-oriented structures, both tree-based and acyclic network-based, has been studied little, and needs a great deal of work.

(4) Some characterization of various data structures which could predict whether they would be best represented by an acyclic data base or a tree-

structured data base would be highly useful in future studies. Perhaps there is a way of setting up parameters to model such characteristics as types of data, richness of connections, depth, bushiness of connections, etc. It would be useful also to know whether a single context should be transformed to a tree by the T-mapping in Appendix B or not, in various cases.

(5) The retrieval expression defined in this paper is simplified to de-emphasize the action dictated by the left-hand side. It would be useful and important to define and possibly standardize the types of operators besides PRINT which could occur in the left-hand side, and the various options which control processing both after and during the retrieval. These operators would most likely include averaging, counting, summing, etc.

There are, no doubt, other areas to be explored with respect to the subject of networks in data management. The question of which applications are most appropriate and of the costs involved will remain unanswered until the tasks above, and possible some others, can be undertaken.

APPENDIX A

S.Y. Bang (B2) presents a general result about a qualification mapping, which applies to acyclic data bases. We assume the existence of a PATH predicate, and a means of discovering whether or not a path, or sequence of edges, exists between any two nodes in the data base. The exact mechanism used is unimportant for Bang's result.

Suppose that node x in typeset T_i satisfies some selection clause, and we wish to determine which datasets in typeset T_j are qualified by their path association with node x. The most straightforward approach to this is to let all the nodes in T_j which are connected to x by some path be qualified. If PATH (x,y) = 1 means there is a path from x to y, we may define a qualification mapping Q from T_i to T_j as

$$Q(i,j,x) = \{ y \in T_j \mid \text{PATH}(x,y) = 1 \},$$

and extend this to a set $X \subseteq T_i$, by the definition

$$Q(i,j,X) = \bigcup_{x \in X} Q(i,j,x).$$

Bang has shown that such a qualification mapping fails

to be a homomorphism because in general

$$Q(i,j,X \cap Y) \neq Q(i,j,x) \cap Q(i,j,Y)$$

and

$$\widetilde{Q(i,j,X)} \neq Q(i,j,\tilde{X}).$$

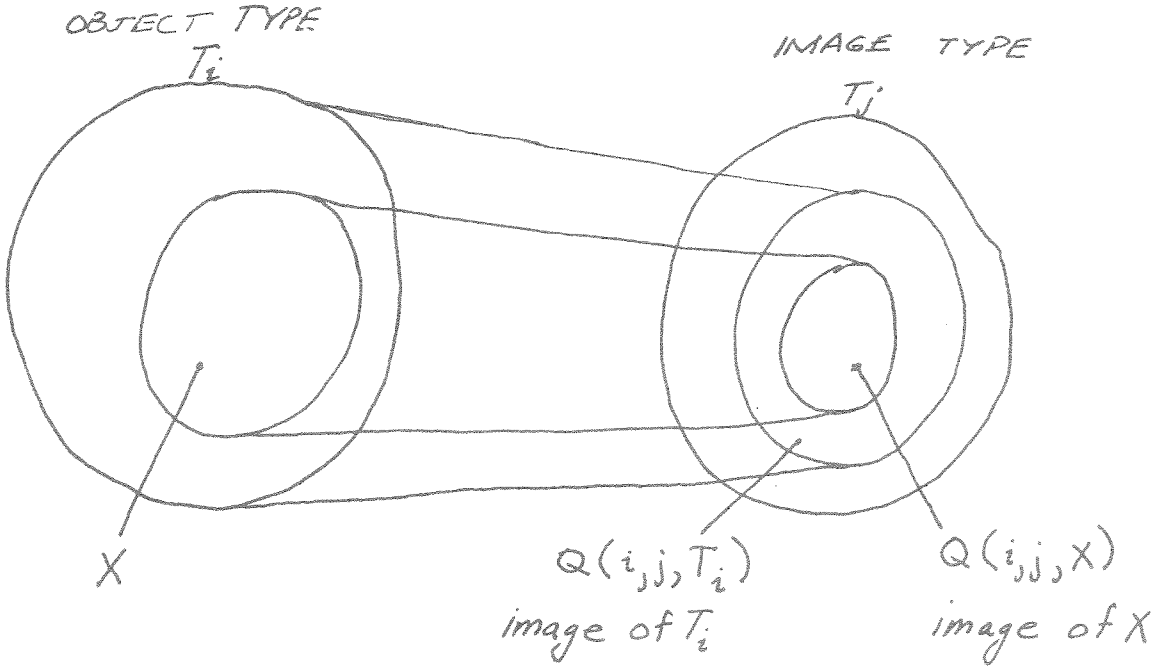
Bang's result is applicable to the acyclic digraph-based data structure, and therefore produces the following semantic problem:

Suppose a is an attribute of type T_j and S1 and S2 are simple conditions of type T_i and Z is a retrieval expression of the form

PRINT a WHERE S1 AND S2

The problem is, in designing the retrieval algorithms for this type of expression, defining the domain of the operator AND in the phrase S1 AND S2. Clearly this must correspond to some intersection of sets of datasets, but how are they to be produced? There are really two choices here. We may find a set in T_i to satisfy S1 and a set in T_i to satisfy S2, use the Q-mapping to find relatives of these sets in type T_j, and then form the intersection of the relatives. Or, we may first satisfy the clause

"S1 AND S2" with an intersection of datasets in T_i, and then find the relatives of that set in T_j. Bang's result assures us that these two methods will, in general, produce different answers. The process of qualifying datasets from selected datasets at other levels must therefore be defined in such a way that no such ambiguity is possible. The mapping Q, where X T_i, from X to T_j, will be referred to as a Q-projection of X into T_j and the set Q(i,j,X) as the image of X in T_j.



Bang (B2) has shown that if $X \subseteq T_i$ then $Q(i,j,X) \subseteq Q(i,j,T_i)$, so the figure above depicts accurately the general Q-projection scheme.

Although Bang does not consider contexts of the acyclic data structure, the indication of a non-homomorphism in the general Q-projection is enough to warrant separating the operations of selection and qualification, as is done in this paper.

APPENDIX B

One solution to the problem of structural representation of the acyclic data base limited to a single context is to map the structure table into a forest of trees. This may prove to be useful in situations for which the system needs a structure table in a form such that (1) a node label may be represented more than once, and (2) each node label is paired with exactly one trace. The trace mapping, or T-mapping, is defined to produce this form from a structure table.

Definition of T-mapping

Suppose Q is a structure table with m types $T_1,\ldots,T_m$, one context $C:T_1\cdot\cdot\cdot T_m$, and k traces. Then each trace in the table is an m-digit integer, with pre-specified fields $f_1,\ldots,f_m$, numbered from left to right. Let $I_1,\ldots,I_m$ each be integers based on the fields $f_1,\ldots,f_m$, respectively. In general because of the multiplicity of traces, the field sizes for the I fields will need to be greater than or equal to the field sizes for the f fields. The example to come shortly will illustrate a case where, in the structure table, only digits up to 9 were sufficient to record the various traces. After the T-mapping, it will be seen that larger digits are

necessary. In terms of computer storage "larger digit" may be translated to "larger field". We assume that the structure table has been sorted into k ordered pairs $(N_1,t_1),\ldots,(N_k,t_k)$, where $t_i < t_{i+1}$, i=1,...,k-1, and each t_i is a trace of N_i. The following is an algorithm which describes the T-mapping for one context:

TM.1. Set m integers $n_1,\ldots,n_m$ to zero.

TM.2. Set j to one.

TM.3. For i = 1,...,m, set $d_i = 0$.

TM.4. Assign the m fields of t_j to the variables $d_1,\ldots,d_m$, respectively.

TM.5. For all i, i = 1,2,...,m,

if $d_i = 0$, then set I_i to zero,

otherwise

if $l_i \neq d_i$ then set I_i to $n_i + 1$ and set n_i to n_{i+1},

otherwise

set I_i to l_i.

TM.6. For i = 1,...,m, set $l_i = d_i$.

TM.7. Form the trace T_j by concatenating the integers $I_1,\ldots,I_m$, and form the ordered pair (N_j,T_j).

TM.8. If j = m then stop. The sequence of pairs $(N_1,T_1),\ldots,(N_k,T_k)$ is the result of the T-mapping.

TM.9. Increment j by one.

TM.10. Go to step TM.4. For a structure table with several contexts, $C_1,\ldots,C_p$, the T-mapping must

be carried for each context, and the resulting sets of T_j traces stored separately.

T-mapping example

The following mapping is derived from Figure 11 and is an example of the T-mapping of that structure table.

j	N_j	t_j	T_j
1	N9	00100	00100
2	N17	00110	00110
3	N18	00120	00120
4	N16	00800	00200
5	N25	00890	00230
6	N31	00896	00231
7	N32	00897	00232
8	N4	01000	01000
9	N10	01200	01300
10	N19	01230	01340
11	N7	04000	02000
12	N14	04600	02400
13	N22	04660	02400
14	N23	04670	02460
15	N1	10000	10000
16	N5	12000	13000

j	N_j	t_j	T_j
17	N10	12200	13500
18	N17	12210	13570
19	N19	12230	13580
20	N11	12300	13600
21	N20	12340	13690
22	N21	12350	136A0
23	N26	12351	136A3
24	N6	13000	14000
25	N12	13400	13700
26	N13	13500	14800
27	N23	13570	14880
28	N24	13580	148C0
29	N29	13584	148C4
30	N30	13585	148C5
31	N2	20000	20000
32	N8	25000	25000
33	N15	25700	25900
34	N24	25780	25900
35	N30	25785	259D6
36	N25	25790	259E0
37	N31	25796	259E7
38	N32	25797	259E8
39	N3	30000	30000

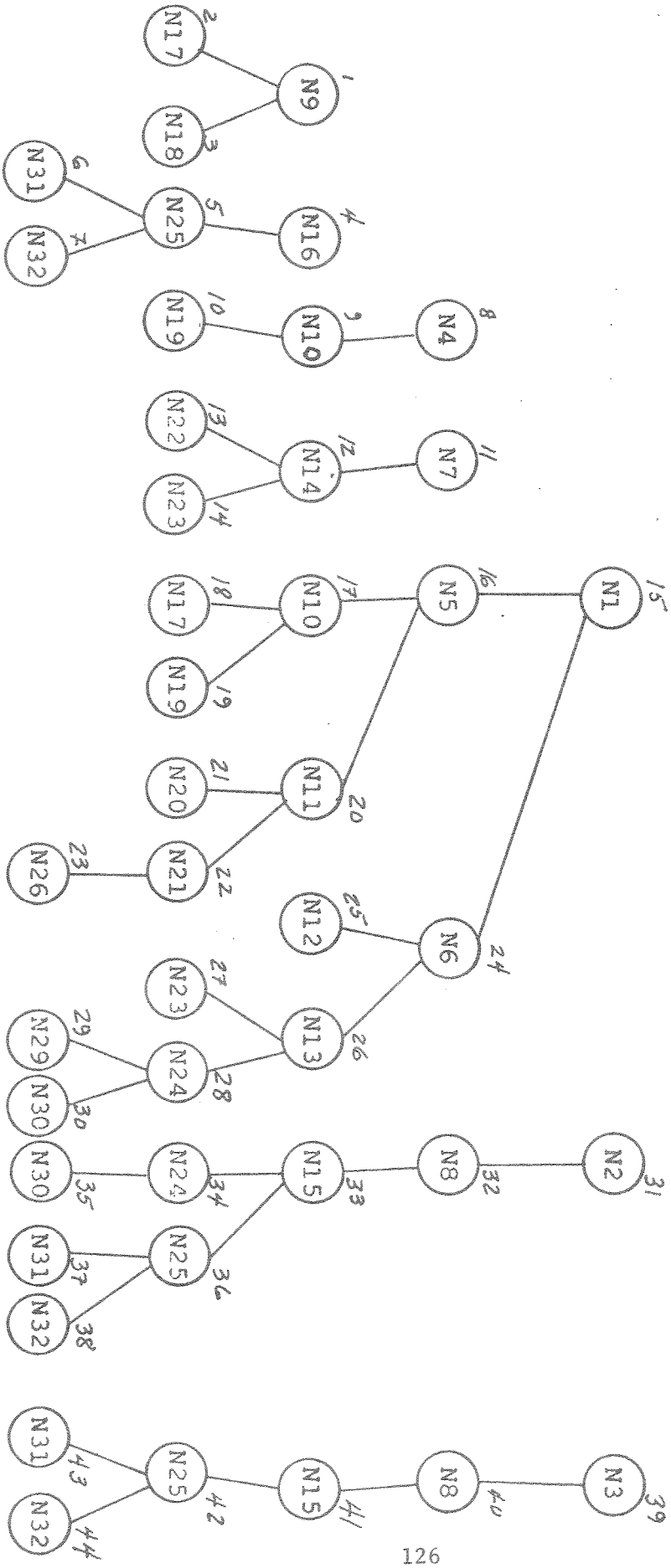

N9
N17
N18
N31
N25
N32
N16
N19
N10
N4
N22
N23
N14
N7
N17
N19
N10
N5
N1
N20
N11
N26
N21
N12
N6
N23
N29
N30
N24
N13
N30
N24
N15
N8
N2
N31
N25
N32
N31
N25
N15
N8
N3
N32

Figure 13

j	N_j	t_j	T_j
40	N8	35000	36000
41	N15	35700	36A00
42	N25	35790	36AF0
43	N31	35796	36AF9
44	N32	35797	36AFA

Note the sequence of digits 1,2,...,9,A,B,...,F. We have resorted to this single character representation of digits for the sake of simple illustration.

The mechanism of generating new digits in algorithm TM produces a different association of node labels and traces, wherein nodes may be redundantly represented. As is shown by Figure 13, diagramming the above example of a structure table, after the T-mapping, produces a digraph which appears to be a "forest" of trees. The structure table of the acyclic digraph, which contains no spurious path information, may be mapped onto a structure table of the sort first specified by Lowenthal (L4). Briefly, Lowenthal has defined a "trace table", in which nodes of a tree are paired one-to-one with a "trace". The trace is an n-tuple which records the linear position of the ancestors of a node, and thus specifies the path to the node via its ancestors. The

"trace", as defined in (L4) however, is based upon the assumption that all nodes are descendants of a unique "TOP" node, unlike the T_j's which are produced by algorithm TM. Because of this difference, the T_j traces produced by algorithm TM will be referred to as T-traces.

A similar T-trace table may be produced by creating a new sequence of nodes in one-to-one correspondence with the T-traces.

Algorithm TM does not produce T-traces with a common digit having different digits to the left, because the condition $l_i \neq d_i$ at step TM.5 compares each new digit to those that were last produced, so that any difference produces an increase in I_i, the digit used in forming the T-trace. Therefore, each node associated with a T-trace has at most one ancestor. In other words, the structure represented by the T-traces is a set of trees, whose roots may be at any level.

BIBLIOGRAPHY

B1 Bachman, Charles, "Introduction to Integrated Data Store," GE Computer Department, Industrial Publication, General Electric Company, P.O. Box 2691, Phoenix, Arizona 85002.

B2 Bang, S. Y., unpublished paper, University of Texas Department of Computer Sciences, Austin, Texas, 1972.

B3 Bleier, R. E., "Treating Hierarchical Data Structures in the SDC Time-shapred Data Management System," Proceedings of the ACM, 22nd Annual Conference, 1967, pp. 41-49.

B4 Bloom, Burton H., "Some Techniques and Trade-offs Affecting Large Data Base Retrieval Times," Proceedings of the ACM, 24th Annual Conference, 1969, p.83.

C1 Childs, D. L., "Description of a Set-theoretic Data Structure," Tech Report 3, Concomp Project, University of Michigan, 1968.

C2 _____, "Feasibility of a Set-theoretic Data Structure," Communications of the ACM, August, 1968.

C3 Clifford, W. D., "A Comparison of the Conventional 3-pointer Data Management System with the Trace System," unpublished paper, Department of Computer Sciences, University of Texas, Austin, Texas, 1971.

C4 CODASYL Data Base Task Group Report, April 1971, Association for Computing Machinery, 1971.

D1 D'Imperio, Mary, "Data Structures and Their Representation in Storage, Parts I and II," NSA Tech Journal, vol. IX, nos. 3 & 4, 1964.

D2 Dixon, P. J., and Sable, J., "DM-1, A Generalized DMS," AFIPS Conference Proceedings, Spring Joint Computer Conference, vol. 30, pp. 185-198, Thompson Books, Washington, D.C., 1968.

E1 Everett, G. E., Data Structure and Algorithmic Determinacy: A Formal Model for the Comparison of Data Management Systems, dissertation,

Department of Computer Sciences, University of Texas, Austin, Texas, 1971.

H1 Harary, F., Graph Theory, Addison-Wesley, Reading, Massachusetts, 1968.

H2 Hardgrave, W. T., Theoretical Aspects of Boolean Operations on Tree Structures and Implications for Generalized Data Management, dissertation Department of Computer Sciences, University of Texas, Austin, Texas, 1972.

H3 Hsiao, D., and Harary, F., "A Formal System for Information Retrieval from Files," Communications of the ACM, February, 1970, pp. 67-73.

I1 IDS/COBOL, GE Information System, CPB-144, General Electric Corporation, August, 1966.

K1 Katzan, Harry, Jr., "Storage Hierarchy Systems," AFIPS Conference Proceedings, Spring Joint Computer Conference, 1971, vol. 38, p. 325.

L1 Landamer, W. I., "The Balanced Tree and Its Utilization in Information Retrieval," Transactions on Electronic Computers, IEEE Journal, vol. EC-xii, no. 5, Dec. 1963.

L2 LePape, Brice, "Study of a Descriptive Language for Statistical Data," Osiris Project, Statistical Office, The European Communities, Brussels, September, 1970.

L3 Levien, R., and Marm, M. E., "Relational Data File: A Tool for Mechanized Inference Execution and Data Retrieval," publ. no. RM-4793-PR, The RAND Corporation, Santa Monica, California, December, 1965.

L4 Lowenthal, E. I., A Functional Approach to the Design of Storage Structures for Generalized Data Management Systems, dissertation, Department of Computer Sciences, University of Texas, Austin, Texas, 1971.

L5 Lum, V. Y., "Multi-attribute Retrieval with Combined Indexes," Communications of the ACM, vol. 13, no. 11, November 1970, pp. 660-665.

M1 McLuskey, William A., "On Automatic Design of Data Organization," AFIPS Conference Proceedings,

Fall Joint Computer Conference, 1970, p. 187.

M2 Minsky, M., Semantic Information Processing, MIT Press, Cambridge, Massachusetts, 1968.

P1 Pratt, T. W., and Friedman, D. P., "A Language Extension for Graph Processing and its Formal Semantics," Communications of the ACM, vol. 14, no. 7, July 1971.

Q1 Quattlebaum, M. V., Subject Headings, Library of Congress, Seventh Edition, Washington, D.C., 1966.

R1 Ramamoorthy, C. V., "Analysis of Graphs by Connectivity Considerations," Journal of the ACM, vol. 13, no. 2, April 1966, pp. 211-222.

S1 Salton, G., Automatic Information Organization and Retrieval, McGraw-Hill Book Company, New York, New York, 1968.