

SCHEMA AND OCCURRENCE STRUCTURE
TRANSFORMATIONS IN HIERARCHICAL SYSTEMS

by

A.G. Dale

N.B. Dale

TR-55

May, 1976

This paper is being presented at the ACM-SIGMOD 1976 International Conference on Management of Data

Technical Report No.55
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712

Introduction

An investigation of transformations of schema and occurrence structures that preserve data independence is of interest from several viewpoints. In the context of data base restructuring, it may be desirable to modify the main schema and its associated occurrence structure from time to time to reflect a changing conceptual view of the phenomena being modelled in the data base. In the context of an application against a given data base, it may be convenient for a user to invoke an external schema that differs from the principal conceptual schema, in which case it is important to ensure compatibility between the external schema and the occurrence structure. In the context of data base optimization it may be desirable to map the occurrence structure (internal schema) to a different state. In all cases it is important that the host system exhibit data independence. That is, if the data base is restructured in the sense that the main schema and the associated occurrence structure are modified, it is desirable that processes defined against the original data base will execute successfully without modification on the restructured data base. The present paper is concerned with this restructuring problem, but considerations regarding restructuring are applicable to other problems of conceptual schema - external schema - internal schema interaction.

In a recent paper, Navathe and Fry [1] have characterized three types of modification for tree-structured schemas, namely (1) renaming of schema constructs, (2) combining of source schema

constructs, and (3) relating of schema constructs. Their paper also explores the consequences of schema modification upon associated logical instance structures, in terms of instance operations and value operations. However, Navathe and Fry do not consider in detail the reasons for imposing restrictions on schema rearrangements, beyond noting that they must be confined to operations on group schemas that lie along a continuous hierarchical path. This notion is related to the observation that valid coverings for accessing operations in hierarchically structured data bases must be confluent hierarchies (Collmeyer [2]; Lavalee, Ohayon and Sauvain [3]).

The purpose of the present paper is to consider the rearrangement rules for tree structured schema and the rules for mapping occurrence structures such that the transformations are query preserving, in the sense that query processes defined on the original schema and occurrence structure will still be valid when executed against a transformed schema and occurrence structure.

More precisely, we wish to consider the problem in the following context:

A hierarchical data base, D , is a pair (S, O) , where S is a schema, and O is an instance network corresponding to S . S is a tree of record (group) types, and each node in the tree corresponds to a record (group) type. Nodes at a given level are unordered.

O is a transitive hierarchical acyclic network of record (group) occurrences. Nodes are partitioned by level and by type,

corresponding to the level and type partitioning defined on S. The network is fully transitive, and no cycles are permitted. A special case of such a network is a tree structure.

The data model, as noted in more detail later, incorporates the notion of nodal equivalence. Two nodes, x_1 and x_2 , in the instance network are data equivalent if x_1 and x_2 are of the same node type and have identical data values associated with them. Consequently the data model is a network with (possibly) data equivalent nodes.

A hierarchical predicate of P of D is a Boolean expression defined on data element names and values, and for the binary operators (AND,OR) is of the general form:
<data element name> <relational operator> <value> <Boolean operator>
<data element name> <relational operator> <value> . For the unary operator (NOT) the expression is of the form <Boolean operator><data element name> <relational operator> <value> .

A selection set, $S(P,O)$ is defined to be the set of nodes of O selected by P.

Hardgrave [4] identified two systems for deriving selection sets from tree structured data bases, which he characterized as set algebraic and tree algebraic systems. The differences between the two systems are noted by Parsons, Dale and Yurkanan [5,6]. Ray [7] demonstrated that the two procedures could also be defined on data bases organized as hierarchical acyclic digraphs.

In this paper we take the selection process to be a tree algebra. In brief, in this algebra;

1. Predicates are definable only if all the terms in the

expression are within the same family (see data model definitions below).

2. Complementation is interpreted with respect to a local universe consisting of nodes of a given type. If T is a set of nodes of record (group) type R , and N is a set of nodes, $N \subseteq T$, selected by a term of a predicate, then $\sim N = T - N$.
3. Intersection and union operations are realized by defining the operation on normalized sets of nodes as follows: if term 1 of P selects a set, T_1 , of nodes of record (group) type R_1 and term 2 of P selects a set, T_2 , of nodes of record (group) type R_2 , and $\text{Level}(R_1) < \text{Level}(R_2)$, then the operation is defined with respect to T_1 , and a set of nodes, T_3 , such that $T_3 \subseteq \text{Anc}(T_2)$ and T_3 contains nodes only of type R_1 (see data model definitions below).

We can now state the schema and occurrence structure transformation problem formally:

Given two data bases:

$$D_1 = (S_1, O_1)$$

$$D_2 = (S_2, O_2)$$

D_1 is query homomorphic to D_2 if:

1. Every node in S_1 is in S_2
2. Every node in O_1 is in O_2
3. $S(P, O_1) \subseteq S(P, O_2)$ for any P defined over D_1 .

An allowable transformation of D_1 is a transformation that preserves query homomorphism.

The notion of a query homomorphism is of importance in a number of application contexts, as noted previously. In the present paper we consider specifically the case where S_2 is a new conceptual schema for a data base for which S_1 is the current conceptual schema, and we wish to preserve the integrity of hierarchical predicates defined on D_1 in the sense that the selection sets produced in D_1 are also produced in D_2 .

It is clear that a query homomorphism requires a mapping between O_1 and O_2 such that the stated condition on the selection sets is met. In particular, a mapping R must exist, such that given $D_1 = (S_1, O_1)$, D_1 is query homomorphic to $D_2 = (S_2, R(O_1))$.

The remainder of the paper will be devoted to a more formal description of the data model and to a description of a mapping that will preserve query homomorphism.

Data Model

Definitions

Level: Given a tree, T , if t_0 is the root of T , $\text{Level}(t_0) = 0$.

If $t_1, t_2 \in T$, $\text{Level}(t_1) = k$, and t_2 is a child of t_1 , then $\text{Level}(t_2) = k + 1$. The level of a record (group) occurrence is the level of the record (group) in the schema tree.

Descendant set: Given a tree, T , and a node $t_i \in T$, $\text{Desc}(t_i) = \{t \in T \mid t \text{ is a node in a tree with root } t_i\}$.

Given an acyclic digraph, (N,E) , where N is the set of nodes and E is the set of edges, a partitioning by level on N , and a node $n_i \in N$, $\text{Desc}(n_i) = \{n \in N \mid n \text{ is a node of a subgraph with vertex } n_i \text{ and Level}(n) > \text{Level}(n_i) \text{ for all } n\}$.

Ancestor set: Given a tree, T , and a node $t_i \in T$, $\text{Anc}(t_i) = \{t \in T \mid t_i \in \text{Desc}(t)\} - \{t_i\}$

Given an acyclic digraph, (N,E) partitioned by level, and a node $n_i \in N$, $\text{Anc}(N_i) = \{n \in N \mid n_i \in \text{Desc}(n)\} - \{n_i\}$

Simple broom set: Given a tree, T , and a node $t_i \in T$, the simple broom set of t_i , denoted $B(t_i) = \text{Desc}(t_i) \cup \text{Anc}(t_i)$. Similarly, given an acyclic digraph, (N,E) partitioned by level, and a node $n_i \in N$, $B(n_i) = \text{Desc}(n_i) \cup \text{Anc}(n_i)$

Data base: A data base D is a pair (S,O) where S is a tree structured schema and O is an acyclic digraph such that each node $n \in O$ represents a record (group) occurrence.

Node type: If X is a record type in a schema S , a node $p \in O$ which represents an instance (occurrence) of X is said to be a node of type X .

Data equivalent set: If there exists a set of nodes $M \in O$ such that each member of the set is of the same node type and represents identical record (group) instances, the set M is termed a data equivalent set.

Family: If S is a tree-structured schema, with a terminal node s_i , a family $F_i \in S$, is $B(s_i)$. If there are j terminal nodes, there are j families in the schema.

Comment on the Data Model

The data model introduces a new basic construct, namely the data equivalent set. The utility of the construct will be seen in the discussion of occurrence structure mapping.

In brief, a network data model utilizing data equivalent sets, offers two main advantages for implementation purposes. First, it permits query homomorphic mappings that introduce less redundancy in the occurrence structure representation than would be the case if the occurrence structure were a tree. Secondly, it provides for a representation that is more economical than a labeled digraph, which otherwise would be necessary to indicate allowable paths in the occurrence structure.

Allowable Restructuring Operations: Schema Level

In the present section we identify the conditions for an allowable data base restructuring. We assume that a restructuring operation involves (1) a schema transformation and (2) a corresponding transformation of the occurrence structure.

With respect to schema transformations, we restrict consideration to the class of transformations that comprise rearrangements of an existing schema at the record (group) level (no nodes are

added and none are deleted), and exclude the trivial case of record (group) permutation at a given level of the schema.

As noted above, the selection processes invoked by hierarchical predicates are defined over families. Consequently, an allowable schema transformation must preserve family structure.

Let S_1 be the initial state of a tree structured schema

S_2 be the schema in its rearranged state

$B_1(s_i)$ be the simple broom set of s_i in S_1

$B_2(s_i)$ be the simple broom set of s_i in S_2

An allowable schema transformation must satisfy the condition:

$$B_1(s_i) \subseteq B_2(s_i) \text{ for all } s_i \text{ in } S_1, S_2.$$

Clearly, if the condition is satisfied, any family that existed in S_1 will exist in S_2 , at least as a subset of an S_2 family.

We proceed now to investigate in more detail schema rearrangement possibilities which satisfy the stated condition.

Any schema tree rearrangement can be decomposed into a series of simple moves of a record type R . There are two types of such simple moves: type (1) where record type R (level i) is interchanged with its parent, record type S (level $i-1$) and type (2) where record type R (level i) is inserted as the parent of its own parent, record type S (level $i-1$). The children of record type R become the children of record type S in both cases. Although these two steps appear similar, they have different consequences in certain cases.

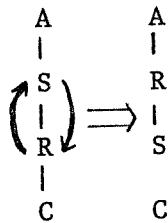
The nodes of any schema tree can be characterized as branching

(two or more children record types) and nonbranching (at most one child record type). Therefore there are four possible cases when applying simple moves: both nodes R and S are nonbranching; node R (the child) is branching and S is not; node S (the parent) is branching and R is not; or both R and S are branching.

The implications of applying the two types of simple moves in each of the four cases to a schema tree are investigated below. A is a record type which is the parent of S; C, D and E are record types which are the children of S or R. Record types A, C, D and E may be branching, nonbranching or null.

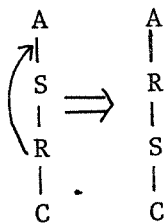
CASE I. Both R and S are nonbranching nodes.

Type (1) move.



It is clear that interchanging two nonbranching nodes will have no effect on the broom sets of either node and therefore is an allowable transformation provided one node is in the broom of the other. This condition is ensured by the definition of the simple moves: a child and its parent are by definition in the same broom.

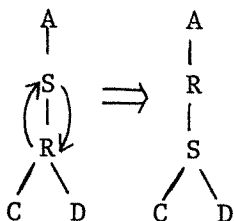
Type (2) move.



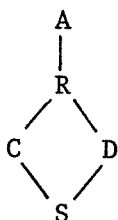
As the diagram shows, a type (1) move and a type (2) move are identical in the case where both R and S are nonbranching nodes.

CASE II. Node R (the child) is branching and node S is nonbranching.

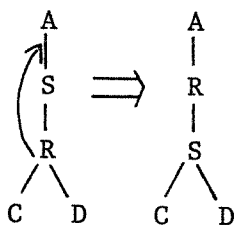
Type (1) move.



The broom sets of each record type are identical in this case and therefore the transformation is allowable. Note that our definition of simple moves eliminates the possible alternative mapping shown below:



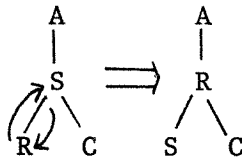
Type (2) move.



Again the diagram shows that a type (1) move and a type (2) move are identical in this case where the child is branching but the parent is not.

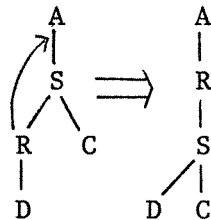
CASE III. Node R (child) is nonbranching and node S (parent) is branching.

Type (1) move.



This transformation is not allowable, because the broom set(s) of the sibling(s) of R will be different in the transformed schema.

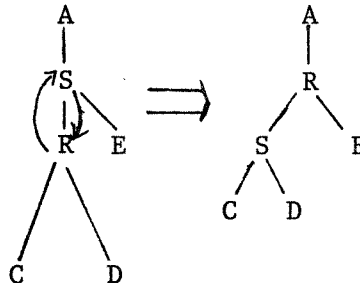
Type (2) move.



In this case the original broom set of the sibling of R will be a subset of its broom in the new schema, and thus this is an allowable schema rearrangement.

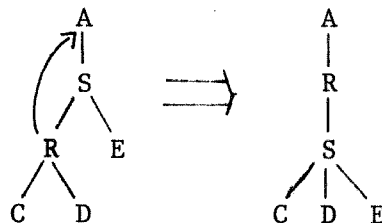
CASE IV. Both nodes R and S are branching.

Type (1) move



This is not allowable for the same reason the type (1) move is not allowable in Case III.

Type (2) move.



The broom of each record type has either been enlarged or not changed. Therefore this transformation is allowable.

In summary, we have identified schema rearrangement operations that satisfy a necessary condition for query homomorphism between two data bases. Any composition of allowable simple rearrangements will generate an allowable final schema state.

Occurrence Structure Mapping

Query homomorphism also requires a mapping from the original occurrence structure to a new occurrence structure corresponding to the new schema such that (a) all nodes in the original structure occur in the new structure and (b) the selection sets produced by a hierarchical predicate P in O_1 are also produced in O_2 (possibly as subsets of $S(P, O_2)$).

The following is a general algorithm for such a mapping.

Given schema 1 and schema 2 where schema 2 is the result of an allowable type (1) or a type (2) move applied to schema 1.

Given the logical occurrence structure defined by schema 1.

Given record types R and S used in the transformation of schema 1 into schema 2.

Assume that for each instance s_j of record type S in the logical occurrence structure defined by schema 1, there are k instances of record type R (r_1, r_2, \dots, r_k).

STEP 1: Make $k-1$ copies of the descendant set of s_j less the union of the descendant sets of all the r 's, in

- the logical occurrence structure defined by schema 1.
- STEP 2: Join the descendant set of each r_l ($l=2,k$) as the right subtree of the $l-1$ copy of s_j made in step 1.
- STEP 3: Remove from the original logical occurrence structure defined by schema 1 all descendant sets of r_l ($l=2,k$).
- STEP 4: Insert the $k-1$ trees created in step 2 into the logical occurrence structure defined by schema 1 as siblings of s_j , i.e. as subtrees of the parent of s_j .
- STEP 5: Change the parent/child relationship so that r_l is the parent of s_j^l and the children of r_l become the children of s_j^l , ($l= 1,k$).
- STEP 6 (optional): Remove redundancies (explained later).
- STEP 7: Repeat steps 1-6 for all s_j in the logical occurrence structure of schema 1.

The logical data structure defined by schema 1 and amended by the above 7 steps is now a logical occurrence structure defined by schema 2. The original plus the copies of a particular node form a data equivalent set. In the following section, the algorithm will be applied to each of the four cases defined previously.

Application of the Algorithm.

- Given: a as an instance of record type A in schema 1.
- r as an instance of record type R in schema 1.
- s as an instance of record type S in schema 1.
- c as an instance of record type C in schema 1.
- d as an instance of record type D in schema 1.

e as an instance of record type E in schema 1.

n as the number of instances of record type A in the logical occurrence structure of schema 1.

m_i as the number of instances of record type S per instance of a_i ($i=1,n$)

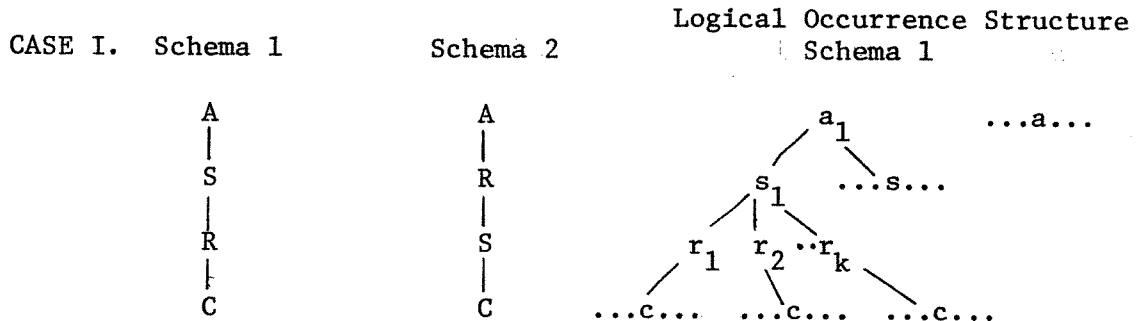
k as the number of instances of record type R per instance of record type S.

Let ... (a,s,r,c,d,e)... represent zero or more occurrences of the corresponding record type at that place in the logical occurrence structure.

Let (r_j) represent the set of subtrees of r_j . (r_j) may be null.

To cover all the s's in the logical occurrence structure, the algorithm should be applied in the following order:

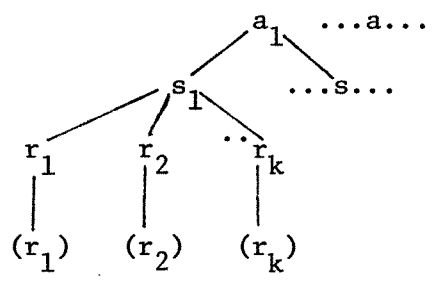
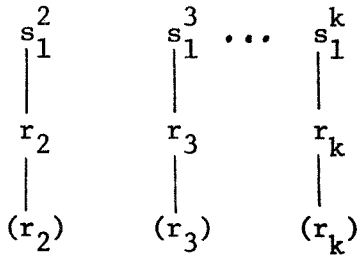
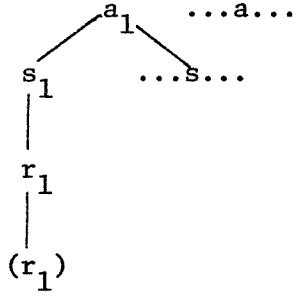
$((a_i, s_j) \quad j=1, m_i) \quad i=1, n$



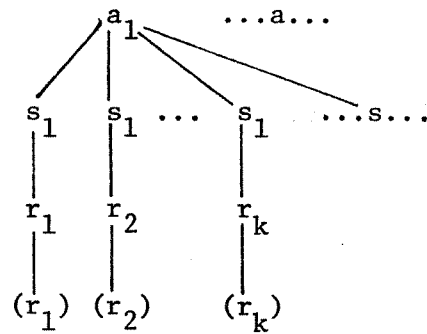
Since the logic of the algorithm is not based on C being a terminal record type, replace ...c... by (r_j) . Table 1 shows the logical occurrence structure under schema 1 and the step by step transformation on the structure which turns it into a logical occurrence structure under schema 2. Where new structure is created it is shown in the right hand side of Table 1.

Table 1

Algorithm applied to Case I

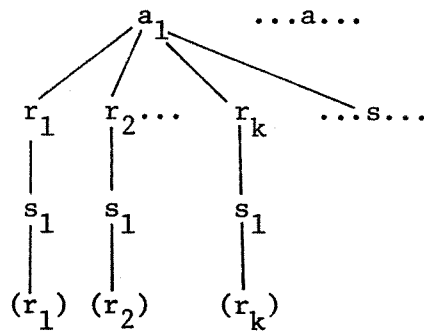
<u>Occurrence Structure</u>	<u>Created Structure</u>
<p>Step 1:</p> 	$s_1^2 \quad s_1^3 \quad \dots \quad s_1^k$ <p>where the superscript refers to the (copy + 1)</p>
<p>Step 2:</p> <p style="text-align: center;">Same as step 1.</p>	
<p>Step 3:</p> 	<p>Same as step 2.</p>

Step 4:

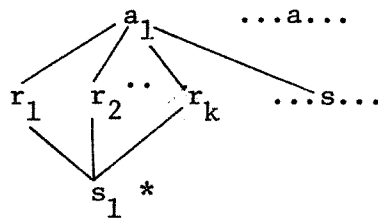


Merged into occurrence structure

Step 5:

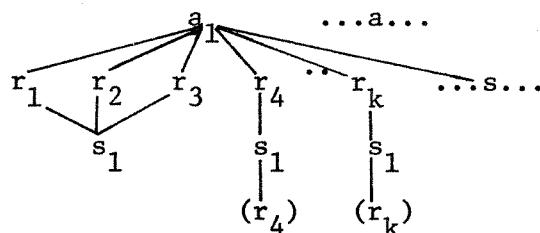


Step 6a:



* This case only applies if all (r_i) are null

Step 6b:



For ease of visual presentation the r_i are numbered so the terminal cases are r_1, r_2 & r_3 .

In step 1 note that the descendant set of s_1 less the descendant sets of r_l ($l=1,k$) is s_1 itself. This will always be the case where record type S is nonbranching.

Step 6 is optional because the logical occurrence structure produced without it is a logical occurrence structure defined by schema 2. It is a tree structure and because of the way it was constructed contains consistent redundancies. If the occurrence structure is required to be a tree, then these redundancies are necessary. However if we permit the occurrence structure to be a network as assumed in our data model, then part of the redundancy can be removed.

A procedure for removing this redundancy has two parts. One is determined by the schema 1 structure itself, and the other depends on the logical occurrence structure under schema 1.

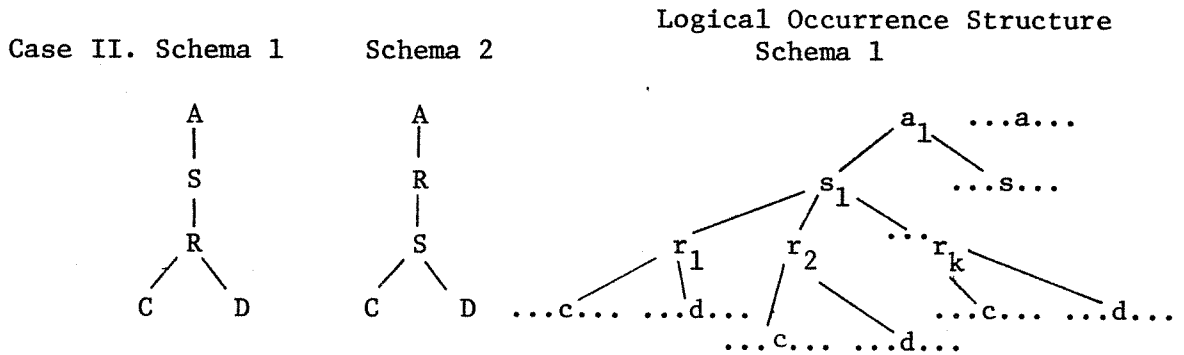
6a. If record type R is a terminal node in the schema 1 tree, steps 1-4 can be omitted and step 5 modified as follows. Change the parent/child relationship so that r_l ($l=1,k$) is the parent of s_j .

As can be seen in Table 1, this creates k in-branches to node s_j .

Even if record type R is nonterminal in the schema 1 structure, any particular r in the occurrence structure may be terminal. If there are two or more r's for any s_j which are terminal, they need point only to s_j instead of as many separate copies as there are terminal r's. This can be stated as follows:

6b. Scan the r_l ($l=1,k$) for terminal r 's. If none are found, no redundancy exists for this s_j . If one is found, say r_p , continue scanning the r_l ($l=p,k$). For each additional terminal r remove s_j^l and make r_l a parent of s_j^p

Examples of 6a and 6b are also in Table 1.



Since (r_l) is defined as the set of subtrees of r_l , and is not limited to subtrees with instances of the same record type as roots, (r_l) can be substituted for $\dots c \dots \dots d \dots$ in the logical occurrence structure of Case II. Hence the application of the algorithm through step 5 is exactly the same for Case I and Case II. However 6a can never be applicable in Case II because record type R is defined as branching and can therefore not be a terminal node in the schema tree. Step 6b is applicable as in Case I.

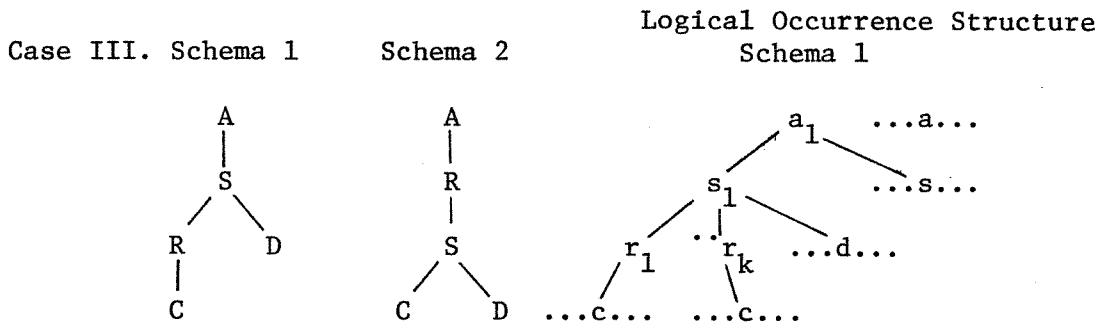
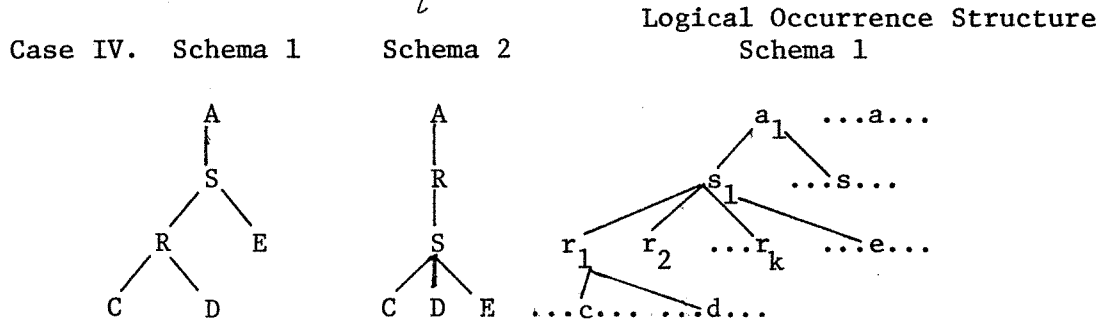


Table 2 shows the application of the algorithm to Case III. Note that 6a and 6b are both applicable. (Note ...c... has again been replaced by (r_2))



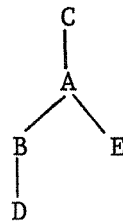
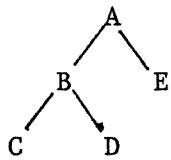
Replace ...c... ...d... by (r_2) and this case reduces to Case III. Note however that 6a is not applicable to this case.

Composition of Operations

If a particular schema rearrangement requires more than one simple move, the algorithm can be applied iteratively. Rather than attempting to prove the correctness of the result in all cases in this paper, a concrete example is given.

Example: Schema 1

Desired Schema



This can be accomplished by 2 type (2) moves (both Case III).

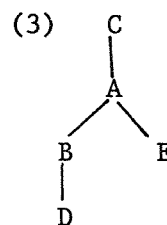
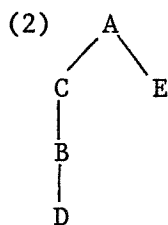
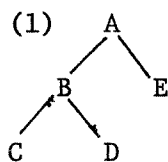
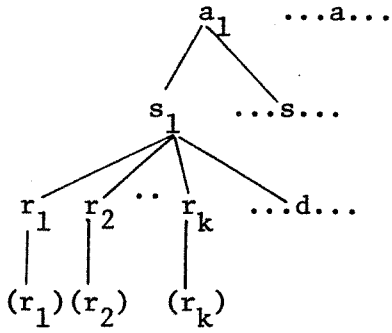


Table 2

Algorithm applied to Case III

Occurrence Structure

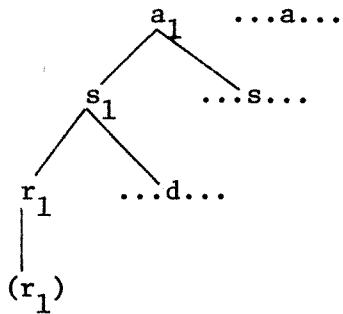
Step 1:



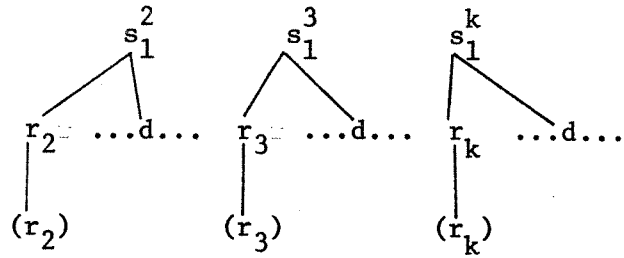
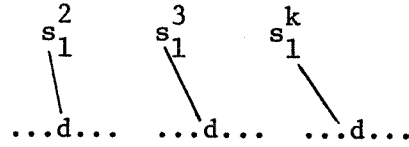
Step 2:

Same as step 1.

Step 3:

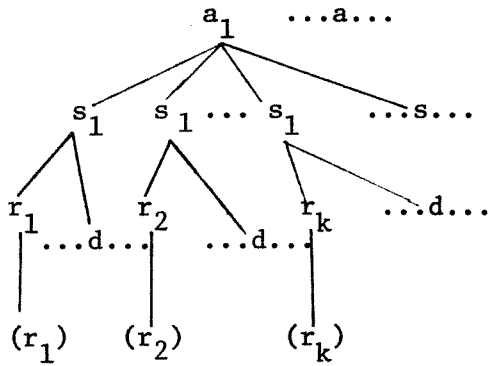


Created Structure

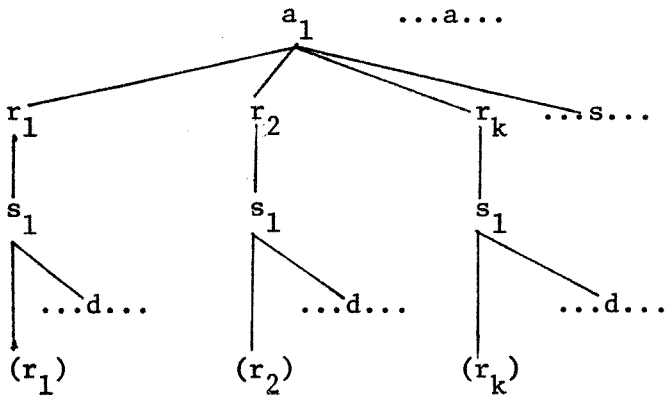


Same as step 2.

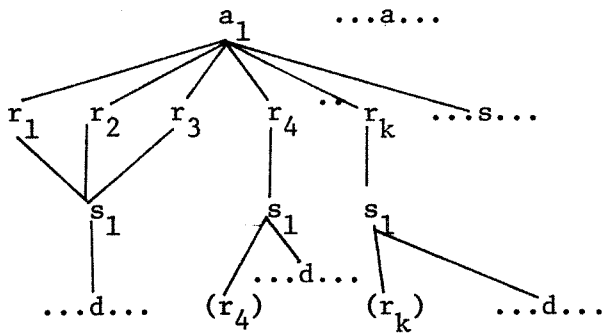
Step 4:



Step 5:



Step 6:

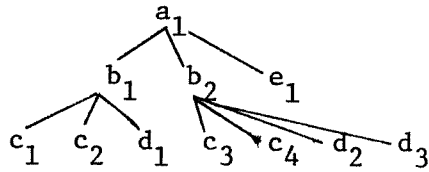


Merged into occurrence structure (1)

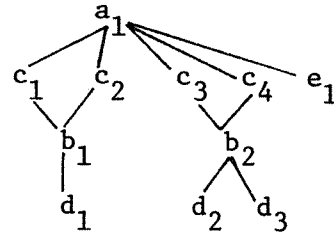
For ease of visual presentation the r_i were numbered so the terminal cases were r_1, r_2 & r_3 .

In the terminology of the algorithm record type R is record type C and record type S is record type B in schema 1.

Logical Occurrence Structure
Schema 1

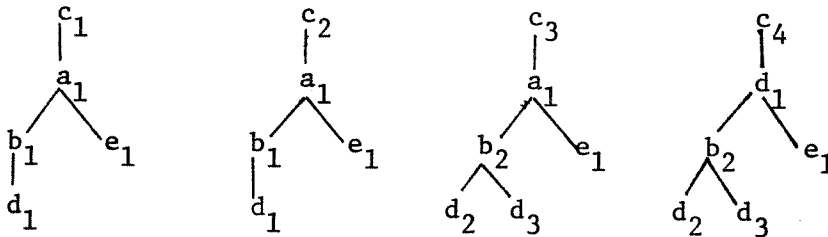


Logical Occurrence Structure
Intermediate Schema



In the second application of the algorithm, record type C corresponds to R and record type A corresponds to S.

Logical Occurrence Structure



It is obvious that redundancy exists which can not be seen until the algorithm has completed operating on all the s_j for a particular a_i . Therefore one additional step is needed on all applications of the algorithm except the first.

6c. When $j = m_i$ in the iteration sequence, compare pair wise the m_i subtrees of a_i . When a duplicate subtree is found, say r_p and r_q , remove the subtree of r_q and make r_q a parent of the subtree of r_p .*

* Note the duplicate subtrees will occur where ever r_p and r_q were the parent of the same child.

Applying this step to the preceding example, the logical occurrence structure becomes



Comment on the Mapping Algorithm

The mapping preserves query homomorphism because the algorithm preserves by construction the identity of original broom sets, and therefore of selection sets, in the mapped structure. That is, every broom set existing in the original structure exists at least as a subset of a broom set in the mapped structure; moreover, the algorithm ensures that no spurious relationships are introduced into the mapped structure.

The utility of a network data model incorporating data equivalent sets is also demonstrated in the last example. A mapped network structure exhibits substantially less redundancy than the alternative mapping to a tree structure, by minimizing the data equivalent sets.

Summary

We have identified allowable rearrangements for tree structured schemas and an occurrence structure mapping that preserves query homomorphism, given a particular selection algebra.

It appears that the approach taken in this paper is applicable to the investigation of other types of permissible modifications of tree structured schemas, including the analog to the domain migration problem. These investigations will be reported in a subsequent paper.

Further areas for productive investigation relate to the problem of preserving the integrity of (structural) updating processes on mapped structures, and, as noted elsewhere in the paper, to questions of conceptual schema - external schema interaction, and to internal schema mapping problems.

Acknowledgment

The authors wish to thank E. I. Lowenthal, MRI Systems Corporation, for constructive comments on earlier drafts of the paper.

References

1. B. Navathe and J. P. Fry, Restructuring for Large Data Bases: Three Levels of Abstraction, Data Translation Project Technical Report 8.1, The University of Michigan, September 1975.
2. A. J. Collmeyer, Implications of Data Independence on the Architecture of Database Management Systems, Proc. 1972 ACM - SIGFIDET Workshop.
3. P. Lavallee, S. Ohayon, and R. Sauvain, Non-Procedural Access to DMS Data Bases, Proc. 19th International XDS Users' Group Meeting, 1972.
4. W. T. Hardgrave, Theoretical Aspects of Boolean Operations on Tree Structures and Implications for Generalized Data Management, University of Texas at Austin Computation Center TSN-26, August 1972.
5. R. G. Parsons, A. G. Dale, C. V. Yurkanan, Data Manipulation Language Requirements for Data base Management Systems, The Computer Journal, vol. 17 No. 2, May 1974, 99-103.
6. R. G. Parsons, A. G. Dale, C. V. Yurkanan, A Structure Processing Sub-Language for Data Base Management, The University of Texas at Austin Computation Center TSN-28, August 1972.
7. F. B. Ray, Directed Graph Structures for Data Base Management; Theory, Storage Structures and Algorithms, The University of Texas at Austin Computation Center TSN-31, December 1972.