ON STORAGE STRUCTURES AND THEIR TRANSFORMATIONS

Peter Scheuermann
Department of Computer Science
State University of New York
Stony Brook, New York 11794

Ben Shneiderman
Computer Science Department
Indiana University
Bloomington, Indiana 47401

TECHNICAL REPORT NO. 30

ON STORAGE STRUCTURES AND THEIR TRANSFORMATIONS

PETER SCHEUERMANN
BEN SHNEIDERMAN

JUNE, 1975

On Storage Structures and Their Transformations

Peter Scheuermann
Department of Computer Science
State University of New York
Stony Brook, New York 11794

Ben Shneiderman
Computer Science Department
Indiana University
Bloomington, Indiana 47401

Abstract

A number of operations which are of importance when dealing with data reorganization and data translation are introduced. We employ a model based on straightforward mathematical tools to describe these operations more formally and to provide an insight to their meaning and application. We recognize that high level primitive operations are necessary to ensure the correctness of storage structure representations.

Introduction

Choosing the best storage structure and access strategy for the implementation of a given logical data structure is an extremely difficult problem. The current trend in data base management system design is to relieve the casual user from the need to this by providing different levels of interface with the system [6]. The actual burden of implementation and encoding detail is placed upon the Data Base Administrator(s), who is also responsible for the reorganization, maintenance and back-up of the data base. The main reason behind this is the desire to achieve a high-level of data independence [10], so that users' programs are "invariant" under physical storage transformations.

Reorganization of a given data base occurs basically due to two reasons: desire to improve the performance of the system, usually estimated in terms of search time and storage cost [7], or the adaptation to a new software/hardware environment [3].

This paper addresses aspects of the reorganization problem, namely the primitive operations involved in the transformation from one storage structure to another. No attempt is made to provide a complete set of such operations, but rather we point out some of the complexities inherent in the problem and the implications for the development of data description and manipulation languages.

Part 1 - Review of the Storage Structure Model

Before we can speak about the operations for dealing with reorganization of data in physical storage, we need a frame of reference. We briefly outline the model presented in greater detail in [5].

We have characterized the mappings of data to physical storage in terms or standard algebraic tools, namely relations. Two types of relations can be used recursively to convey the meanings of the connectivities for arbitrary structures.

A. Pointer Relations

These are defined by the binary relations $P(S,T)$ on the set of data elements (fields or groups of fields) as follows:

$$P(S,T) \text{ means } S \rightarrow T \quad \text{(read " } S \text{ points to } T \text{ ")}$$

i.e., there is a pointer from S to T. Proceeding recursively:

$$P_2 = P$$

$$P_n(S_1,S_2,\ldots,S_n) = P_{n-1}(S_1,S_2,\ldots,S_{n-1}) \ \& \ P_2(S_{n-1},S_n) \quad n > 2$$

e.g., $P_3(S_1,S_2,S_3) = P_2(S_1,S_2) \ \& \ P_2(S_2,S_3)$ implies there is a pointer from S_1, to S_2 and a pointer from S_2 to S_3 .

It is also useful to have another type of pointer relation $\tilde{P}$, to indicate pairs of data elements symmetrically connected:

$$\tilde{P}(S,T) \text{ means } S \rightarrow T \text{ and } T \rightarrow S$$

i.e., there is a pointer from S to T and a pointer from T to S and we similarly generalize:

$$\tilde{P}_n(S_1,S_2,\ldots,S_n) = \tilde{P}_{n-1}(S_1,S_2,\ldots,S_{n-1}) \;\&\; \tilde{P}_2(S_{n-1},S_n)$$

$\tilde{P}_n(S_1,S_2,\ldots,S_n)$ has the obvious meaning: there is a pointer from S_1, to S_2 , from S_2 to S_3 , from S_{n-1} to S_n , and vice-versa.

B. Contiguous Relations

These describe data items which are stored contiguously on the physical device. $C(S_1,S_2)$ means S_1 is stored contiguously to S_2 ; and similarly:

$$C_2 = C$$

$$C_n(S_1,S_2,\ldots,S_n) = C_{n-1}(S_1,\ldots,S_{n-1}) \;\&\; C_2(S_{n-1},S_n) \qquad n > 2$$

The description can be further refined to distinguish between C_n sequential and random access tuples:

(i) the first denoted by $C_n(S_1,S_2,\ldots,S_n)$ specifies that in order to access a given item S_j you have to retrieve first $S_1,S_2,\ldots,S_{j-1}$

(ii) while the latter, denoted by $\tilde{C}_n(S_1,S_2,\ldots,S_n)$ implies that every item S_j can be accessed at random.

To complete our descriptive mechanism we also need a means of representing the actual assignment of data to particular physical devices. For this purpose we introduce an address space function: $AS : D \to I$, where $AS(d)$ identifies a given device, or device extent, i , on which the data element d is stored. For example, if a given data element resides on a disk, then $AS(d) = i$ represents an encoding of its actual positioning in terms of device, cylinder, track or block.

To illustrate the concepts described above, suppose that we deal with hierarchical records, each containing information about a given vendor, the orders received and the items supplied for each order.(Figure 1). These records may be mapped to physical storage using ring structures as illustrated in Figure 2.

In our formalism this corresponds to:

$$P_{n+2}(V,O_1,O_2,O_3,\ldots,O_n,V) \;\&$$
$$\{P_{k_i+2}(O_1,I_{i1},I_{i2},I_{i3},\ldots,I_{ik_i},O_i) \mid i = 1,\ldots,n\}$$

In addition, the implementor has decided to store the root segments (vendor) and the order segments on device extent 1, and the leaf segments (items) on device extent 2:

$$AS(V) = 1, \{AS(O_i) = 1 \mid i = 1,\ldots,n\} \quad ,$$
$$\{AS(I_{ij}) = 2 \mid j = 1,\ldots,k_i\}$$

Part 2 - High Level Primitive Operations

Our goal is to define more formally the meaningful operations for transforming one storage representation into another. Their use is either in reducing the storage cost or in improving the access time for certain transactions performed on the data base. Some of the operations mentioned here are encountered in other papers dealing with the subject [6,9], while the others have been used in an *ad hoc* fashion by many implementors.

A. Fusion and Fission

The conversion from a linked list allocation strategy to a sequential contiguous allocation strategy is called fusion (Figure 3). The inverse operation is called fission.

In our formalism, we define the FUSION operation as follows:

$$\begin{tikzcd} \bar{P}_n(D_1,\dots,D_n) \arrow[d, Rightarrow, "\text{FUSION}"'] \\ \bar{C}_n(D_1,\dots,D_n) \end{tikzcd}$$

where we have adopted the abbreviation: $\bar{P}_n = P_n$ or $\tilde{P}_n$ and $\bar{C}_n = C_n$ or $\tilde{C}_n$.

This transformation is performed when dealing with records which have a fixed number of fields or segments. However, we have not yet conveyed the complete meaning of the accessing mechanism for the D_i elements in the new contiguous relation. This must be done if the system is to keep track of all search paths. We can see easily that if the physical device in question is a tape, only a sequential C_n can be created. Thus it is necessary to do some "device type checking" to eliminate inconsistent representations (here is where the address space function appears on the scene).

B. Extract and Embed

The next transformation, extract, allows us to convert a list structure into a pointer array, as illustrated in Figure 4. After extracting the pointers from the list elements, the first element (sometimes referred to as header or owner) contains pointers to each member or detail element.

Note that the extract operation saves no space, but simplifies processing for the cases in which no complete traversal of the original list is required. Formally we write:

$$\begin{array}{l} \bar{P}_{n+1}(D_0, D_1, D_2, \ldots, D_n) \\ \text{EXTRACT} \Big\Downarrow \\ \{\bar{P}_2(D_0, D_i) \mid 1 \le i \le n\} \end{array}$$

The inverse of this operation is embedding. As harmless as it seems this transformation involves an unseen complexity. One of the desirable features of a "concrete" storage structure is to preserve all the properties of the more abstract data structure [2]. However, when dealing with the encoding of an information space in storage, a number of superimposed levels of representation are present. The user has one "view" of the logical information, and the implementor imposes on this his own access paths and encoding scheme. When the structure of the user's view is preserved at the encoding level, we can say that the mapping is a type-isomorphism to use a term coined by Childs [1], and it is not difficult to see that the actual requirements of the data-manipulation programs are simplified in this case. For example, going back to Figure 4, if the user visualizes his/her records as composed of a segment (D_0) containing some fixed information about a given employee, and a number of segments (D_i) containing information about different jobs held in a company, the pointer array implementation reflects exactly this situation, namely that the D_i's are disjoint [5]. However the embedded structure does not preserve this property, and extra processing is required.

C. <u>Split and Combine</u>

When records have some long, infrequently accessed fields, it may be more efficient to separate these fields out, store them separately, and access them only when needed. For example, in a collection of art object descriptions, the infrequently accessed biographical information on the artist can be maintained separately, thereby reducing the average transfer time for the records. Figure 5a shows this type of "splitting" and the inverse operation, "combining". We define the SPLIT1 operation as:

$$\begin{array}{c} P_n(C_2(D_{11},D_{12}),\ldots,C_2(D_{n1},D_{n2})) \\ \text{SPLIT1} \Big\Downarrow \\ P_n(D_{11},\ldots,D_{n1}) \;\&\; \{P_2(D_{i1},D_{i2}) \mid 1 \le i \le n\} \end{array}$$

and after the operation:

$$\{AS(D_{i1}) = I1 \mid 1 \le i \le n\}$$

$$\{AS(D_{i2}) = I2 \mid 1 \le i \le n\}$$

where I1 and I2 are indexes of separate address spaces.

A second type of splitting is shown in Figure 5b and can be described as:

$$\begin{tikzcd} C_n(C_2(D_{11},D_{12}),\ldots,C_2(D_{n1},D_{n2})) \arrow[d, Rightarrow, "\text{SPLIT2}"'] \\ P_2(C_n(D_{11},\ldots,D_{n1}),C_n(D_{12},\ldots,D_{n2})) \end{tikzcd}$$

and after the operation:

$$\{AS(D_{i1}) = I1 | 1 \le i \le n\}$$

$$\{AS(D_{i2}) = I2 | 1 \le i \le n\}$$

where I1 and I2 are the indexes of separate address spaces. Of course, the data elements, the D's, may have a complex substructure of their own.

D. Factor and Distribute

The factoring operation, first defined in connection with the DIAM model [6], has the purpose of minimizing storage utilization by eliminating redundancy. Going back to the hierarchical records defined in Figure 1, let us consider the case when all the segments contain the same value, $D_{\hat{m}}$, in one of the fields. Obviously, that value can be "factored" out and stored in the header. Similarly, if any k-level segments contain a common value, it can be factored to the common ancestor, a k-1 level segment. We illustrate this factoring operation in Figure 6 and define it as:

$$\begin{tikzcd} P_{k+1}(R_0,R_1,\ldots,R_k) \arrow[d, Rightarrow, "\text{FACTOR}(\hat{m})"'] \\ P'_{k+1}(R'_0,R'_1,\ldots,R'_n) \end{tikzcd}$$

where

$$R_0 = C_n(D_1,\ldots,D_n)$$

$$R_i = C_m(E_1,\ldots,E_m) \quad , \quad 1 \leq i \leq k$$

$$1 \leq \hat{m} \leq k$$

and in the target structure

$$R_0' = C_{n+1}'(D_1',\ldots,D_{n+1}')$$

$$R_i' = C_{m-1}'(E_1',\ldots,E_{m-1}')$$

Part 3 - Conclusions

No attempt has been made to provide a complete set of operations meaningful to the problem of data reorganization. While the actual transformations from one storage scheme to another could be performed with only a small set of low level primitive transformations [9], no guarantees could be given to ensure the preservation of the more abstract properties of the data. An analogy could be drawn to assembly language operations which give us a great deal of freedom, but do not allow type checking or a protective mechanism. The need for more powerful programming languages for data base systems has been stressed only recently [4]. Data manipulation languages which support reorganization operations should have the capability to handle different levels of abstraction, especially if the system has to keep track of all the search paths and eventually has to automatically generate code for the new storage schemes.

The goal of this paper is to suggest the kind of operations that would be meaningful for data translation at an intermediate implementor-oriented level. As such, the operations are below the level proposed in [8] which uses a high level construct, the form, as a basis for the operations. Our operations are a step above the low level generalized data translator-oriented operations proposed in [11].

We hope to pursue more precise definitions of these operations and will investigate the possibility of establishing a "complete" set of translation operations. Other research directions include the study of the "correctness" of the translation and efficient implementation of the operations.

References

1. Childs, D.L. Extended set theory: a formalism for the design, implementation and operation of information systems. Unpublished paper.

2. Hoare, C.A.R. Proof of correctness of data representations. Acta Informatica 1 (1972), 271-281.

3. Fry, J.P., and Jins, D.W. Towards a formulation and definition of data reorganization. Proceedings of the ACM SIGFIDET Conference (1974).

4. Minsky, N. On interaction with data bases. Proceedings of the ACM SIGFIDET Conference (1974).

5. Scheuermann, P., and Heller, J. A view of logical data organization and its mapping to physical storage. Proceedings of the Third Texas Conference on Computing Systems (1974).

6. Senko, M.E.; Altman, E.C.; Astrahan, M.M.; and Fehder, P.L. Data structures and accessing in data-base systems. IBM Systems Journal 12, 1 (1973).

7. Shneiderman, B. Optimum data base reorganization points. Comm. ACM 16, 6 (June, 1973).

8. Shu, N.C.; Housel, B.C.; and Lum, V.Y. CONVERT: a high level translation definition language for data conversion. IBM Research Report RJ1515.

9. Smith, D. Alternative approaches to the description of the encoding of factored information. SDDTTG Working Paper (1974).

10. Stonebraker, M. Functional view of data independence. Proceedings of the ACM SIGFIDET Conference (1974).

11. Stored data definition and translation task group report.
Draft (April, 1975). To appear.

Typed by Christopher Charles

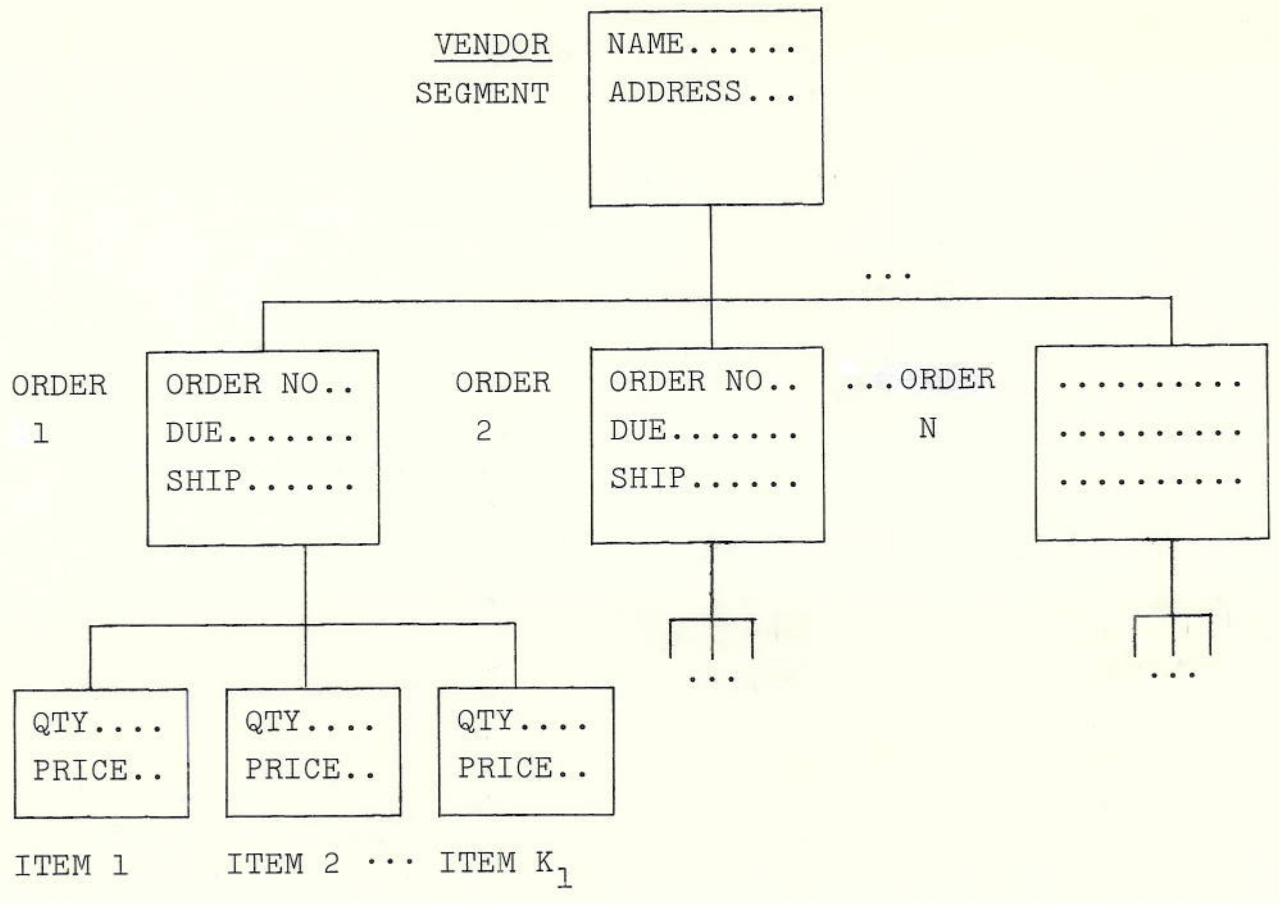
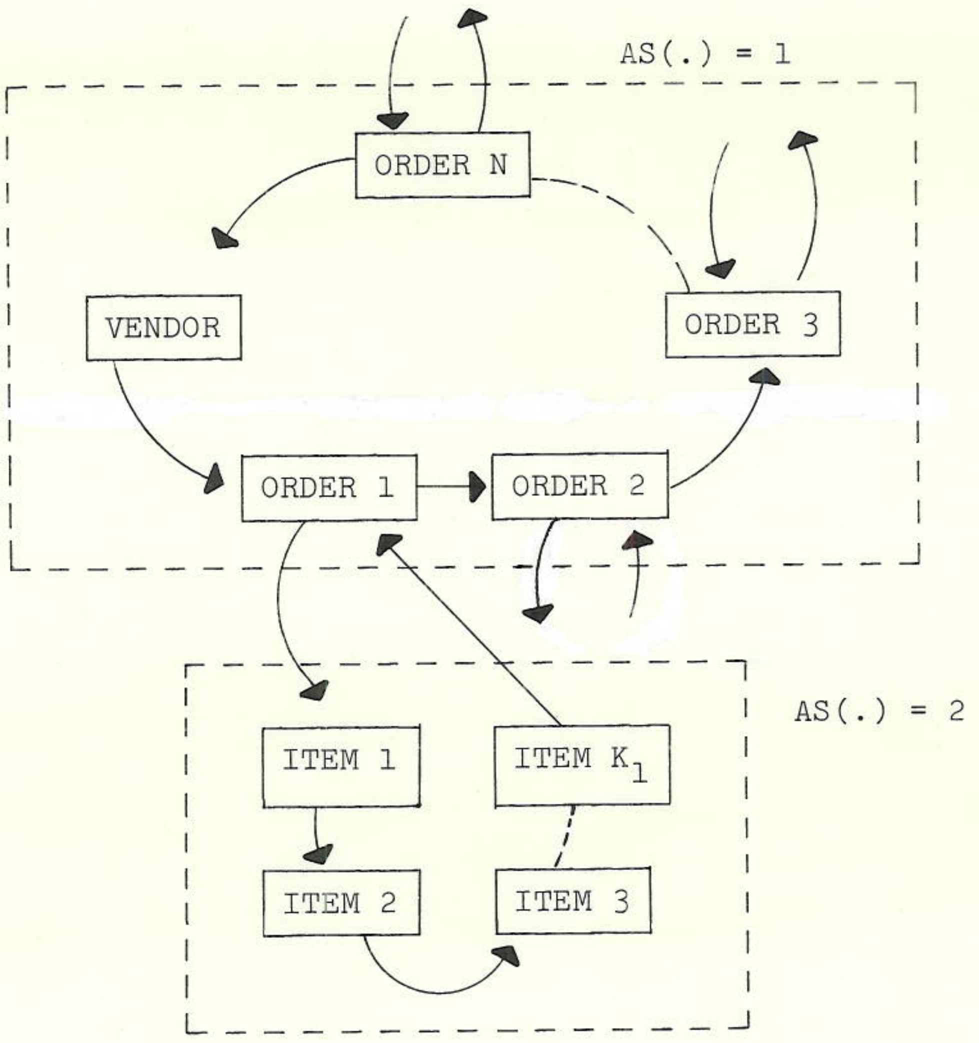


Figure 1 - Logical view of VENDOR/ORDER/ITEMS RECORDS



<u>Figure 2</u> - Physical implementation of VENDOR/ORDER/ITEMS RECORDS

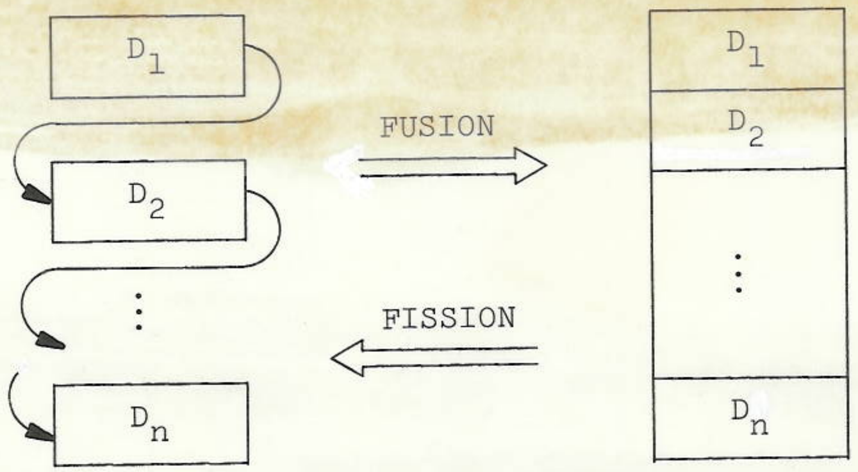


Figure 3

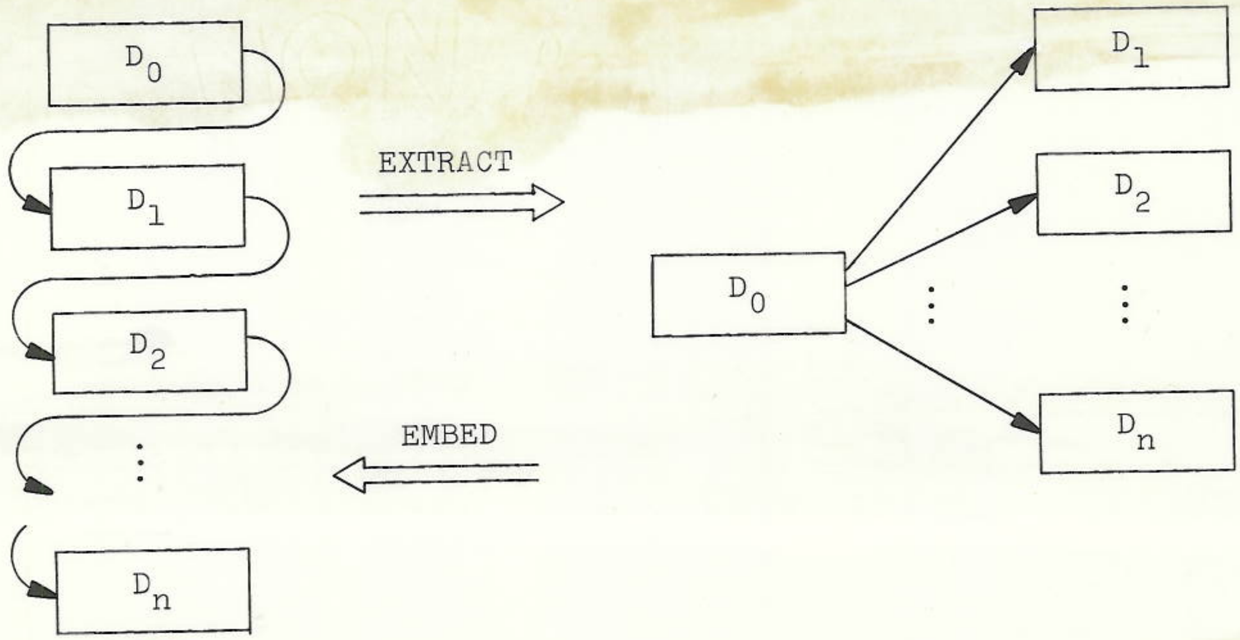


Figure 4

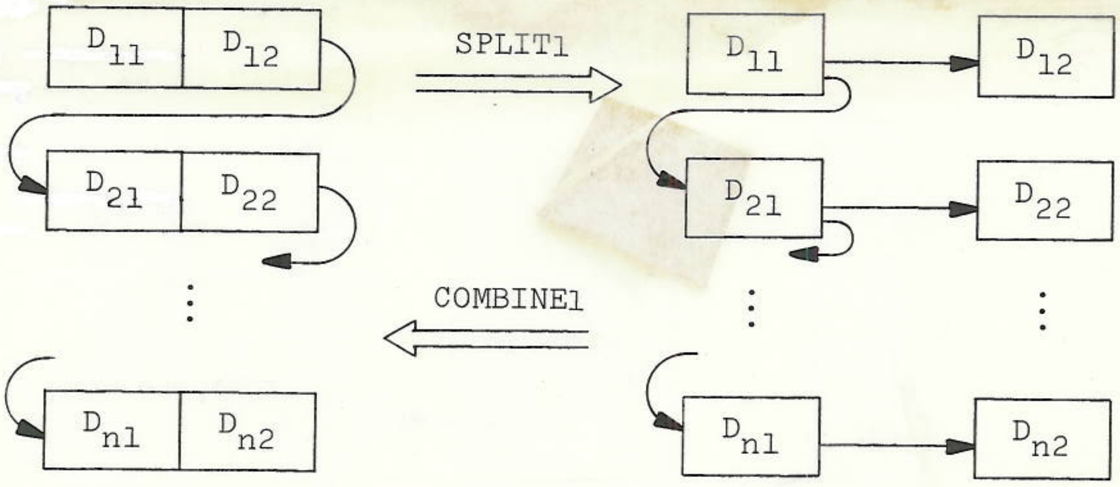


Figure 5a

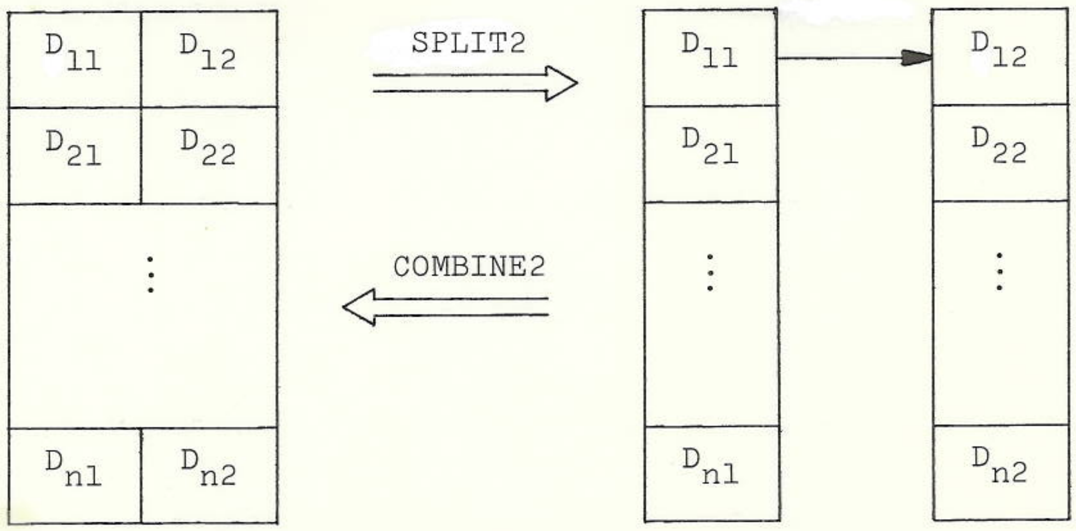
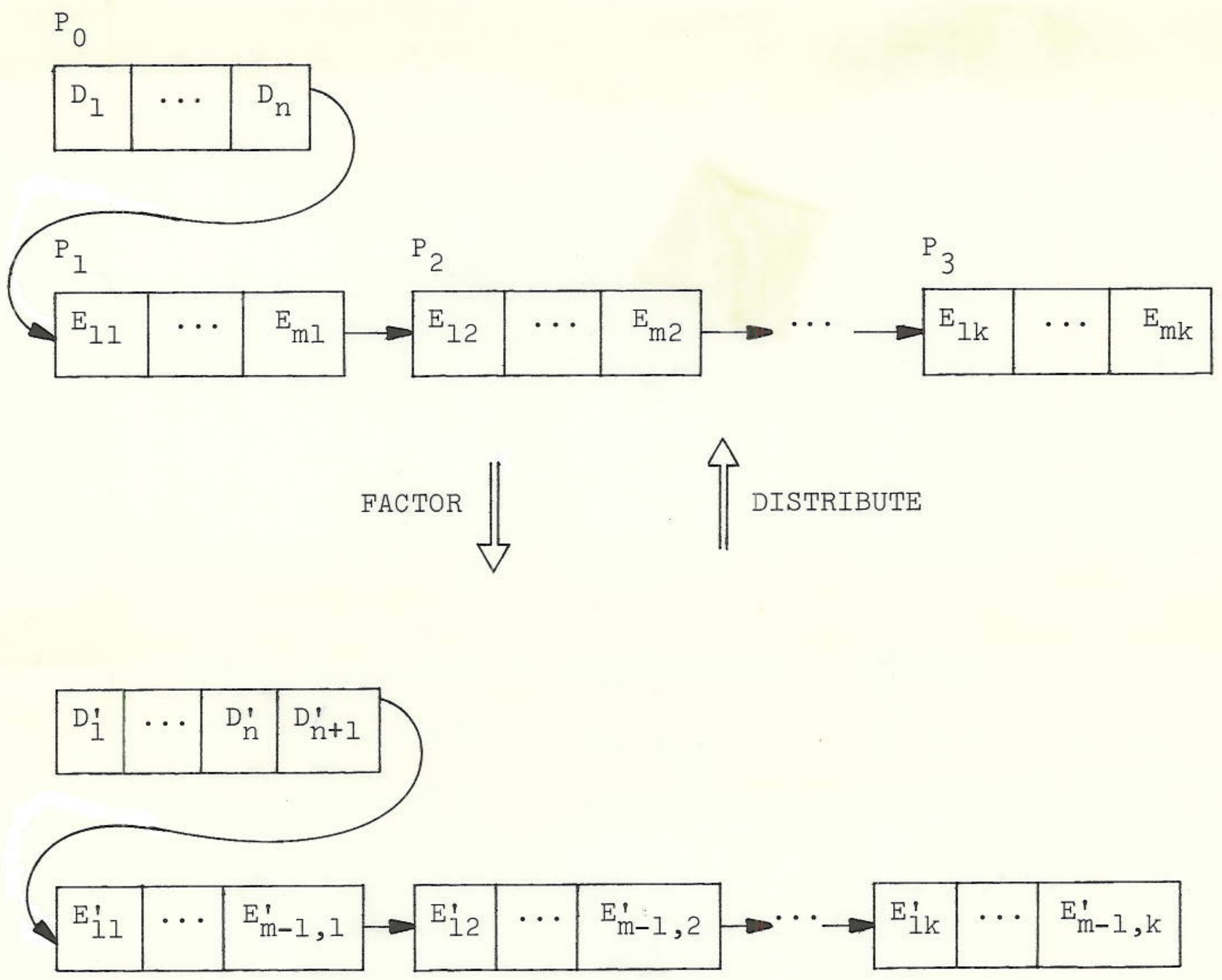


Figure 5b



P_0
D_1 ... D_n
P_1
E_{11} ... E_{m1}
P_2
E_{12} ... E_{m2}
...
P_3
E_{1k} ... E_{mk}
FACTOR
DISTRIBUTE
D'_1 ... D'_n D'_{n+1}
E'_{11} ... $E'_{m-1,1}$
E'_{12} ... $E'_{m-1,2}$
E'_{1k} ... $E'_{m-1,k}$

Figure 6