Vertical Partitioning Algorithms for Database Design

SHAMKANT NAVATHE, STEFANO CERI, GIO WIEDERHOLD, AND JINGLIE DOU
Stanford University

This paper addresses the vertical partitioning of a set of logical records or a relation into fragments. The rationale behind vertical partitioning is to produce fragments, groups of attribute columns, that "closely match" the requirements of transactions.

Vertical partitioning is applied in three contexts: a database stored on devices of a single type, a database stored in different memory levels, and a distributed database. In a two-level memory hierarchy, most transactions should be processed using the fragments in primary memory. In distributed databases, fragment allocation should maximize the amount of local transaction processing.

Fragments may be nonoverlapping or overlapping. A two-phase approach for the determination of fragments is proposed; in the first phase, the design is driven by empirical objective functions which do not require specific cost information. The second phase performs cost optimization by incorporating the knowledge of a specific application environment. The algorithms presented in this paper have been implemented, and examples of their actual use are shown.

1. INTRODUCTION

Partitioning in database design is the process of assigning a logical object (relation) from the logical schema of the database to several physical objects (files) in a stored database. *Vertical partitioning* subdivides attributes into groups and assigns each group to a physical object. *Horizontal partitioning* subdivides object instances (tuples) into groups, all having the same attributes of the original object. We refer to the physical objects that are a result of vertical or horizontal partitioning as *horizontal* or *vertical fragments*.

This paper considers the vertical partitioning problem. We use the term *vertical partitioning* somewhat loosely, in the sense that we allow fragments to overlap, so that partitions are not necessarily disjoint. The term *clusters* has been used in the literature (e.g., [11]) to refer to nondisjoint fragments; but for the sake of convenience we shall use the term *partitioning* to refer to both the disjoint and nondisjoint cases.

This work was performed at Stanford University as part of the Knowledge Base Management Systems Project, supported by the Defense Advanced Research Projects Agency contract N39-80-G-0132. S. Ceri's work was supported in part by a grant from the Italian National Research Council.
Authors' addresses: S. Navathe and S. Ceri, Dipartimento di Elettronica, Politecnico di Milano, Milan, Italy; G. Wiederhold, Dept. of Computer Science, Stanford University, Stanford, CA 94305; J. Dou, Dept. of Computer and Information Sciences, University of Florida, Gainesville, FL 32611.

Vertical partitioning is used during the design of a database to improve the performance of transactions: fragments consist of smaller records, and therefore fewer pages in secondary memory are accessed to process a transaction. When allocating data to a memory hierarchy, vertical partitioning is used to store the attributes that are most heavily accessed in the fastest memory. In the design of a multiple-site distributed database, fragments are allocated, and possibly replicated, at the various sites.

Vertical fragments are ultimately stored in the database by using some physical file structure; the actual implementation of storage structures for fragments is not discussed in this paper.

In order to obtain improved performance, fragments must be "closely matched" to the requirements of the transactions. The ideal case occurs when each transaction "matches" a fragment, because the transaction has to access that fragment only. If certain sets of attributes are always processed together by transactions, the design process is trivial. But in real-life applications one rarely comes across such trivial examples; hence, for objects with tens of attributes, we need to develop a systematic approach to vertical partitioning. As pointed out in [11], an object with m attributes can be partitioned into $B(m)$ different ways, where $B(m)$ is the mth Bell number; for large m, $B(m)$ approaches m^m; for $m = 15$, it is $\approx 10^9$, for $m = 30$, it is $\approx 10^{23}$.

Not may previous attempts to solve this problem have been reported, and none of them has considered all the possible applications of vertical partitioning. The main contributions of our paper are in the following areas:

(i) We extend the work of Hoffer and Severance [13]. They defined an algorithm in which attributes of an object were permuted in such a way that attributes with "high affinity" were clustered together. Affinity among attributes expresses the extent to which they are used together in processing. Starting from the algorithm in [13], several algorithms are developed in this paper for determining the grouping of attributes into nonoverlapping or overlapping fragments.

(ii) We consider the application of partitioning to databases that use one level of memory or a memory hierarchy. We also consider the application of vertical partitioning to distributed databases by determining the allocation of fragments to different sites, allowing nonreplicated or replicated fragments. Thus, we apply vertical partitioning to a much more general environment in comparison to the previous work.

(iii) A unique feature of our approach is the two-step design of fragments. During the first step, vertical partitioning is governed by an empirical notion of partitioning, where one expects that the need for a transaction to visit multiple fragments should be minimized; this step does not require detailed cost factors. During the second step, the design of fragments can be further refined by incorporating estimated cost factors, which reflect the physical environment in which the fragments are eventually stored. Cost factors are based on "logical" performance measures; such measures are not tied to particular access methods or transaction processing strategies.

Our computational experience with the vertical partitioning algorithms has been very encouraging. In this paper we present examples of the partitioning of objects in various ways, using different algorithms under different application environments. Results are supported by intuitive reasoning, to the extent possible.

1.1 Problem Description

We refer to a logical record or a relation as an *object* to be partitioned; each object consists of a set of attributes. An object can be vertically partitioned by considering its projections onto different sets of attributes as new objects, called *fragments*.

A *binary vertical partitioning* (BVP) generates two fragments. A BVP is said to be *nonoverlapping* if the intersection of the two fragments is empty, otherwise it is said to be *overlapping*.

The following is a general statement of the *vertical partitioning problem* (VPP):

Database and Transaction Specification. A database consisting of a set of objects is given in terms of a schema definition. Each object has a set of attributes with given lengths. A set of transactions is defined a priori for the database; our assumption of knowing important transactions a priori is consistent with the common practice of distributed applications and with the notion of precompiled transactions of the R* Distributed Database System [24], or of transaction classes in SDD-1 [19]. The transaction specification relevant to the vertical partitioning of an object consists of the following information:

(i) the frequency (per unit time period) of occurrence of the transaction; when the same transaction can be issued at multiple sites in a distributed database, the frequency at each site is specified;
(ii) the subset of the attributes used for retrieval or update by the transaction;
(iii) the total number of instances of the object selected on the average by one occurrence of the transaction.

This information is typically available after the "logical design" phase of the overall database design activity (see [14, 22] and [23] for a discussion of the overall database design process).

Approach. To place the VPP problem, as defined in this paper, in proper perspective, one may refer to three kinds of decisions:

(i) logical decisions, concerning the structure and composition of the fragments of an object;
(ii) distribution and allocation decisions, concerning the placement of fragments at various storage devices or sites;
(iii) physical decisions, concerning the storage structure (file organization) and access methods for each fragment.

The use of the terms *logical* and *physical* above is only for convenience: it is difficult to argue where the exact boundary lies between these two types of decisions. This paper concentrates on the logical and distribution decisions, assuming that the physical decisions will be taken in a later phase of the design. This, however, does allow the designer to use some "a priori" knowledge about physical cost factors in performing the above two decisions.

Figure 1 shows the overall approach to the VPP problem in this paper. Vertical partitioning will be addressed in two phases:

(1) Empirical design of fragments based on the specified logical access frequencies of the transactions.

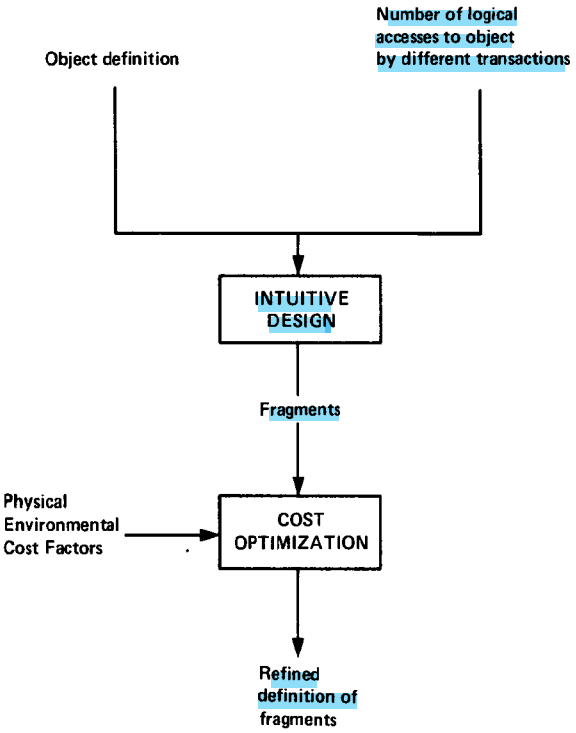

Fig. 1. An overall approach to solving the vertical partitioning problem (VPP).

(2) Design based on cost factors related to the physical environment; this produces a refinement of fragmentation and, if applicable, the allocation of fragments to multiple levels of a memory hierarchy or to multiple sites of a distributed database.

Phase 1 allows the designer to make "reasonable" decisions in the absence of detailed information about the cost factors; of course, phase 1 can be skipped if this cost information is available at the beginning of the design. The early determination of a "reasonable" fragmentation can be used for determining cost factors experimentally, which in turn allows us to refine the design in phase 2. Finally, in some cases, no adequate cost measures are available, and the first approach is the only feasible one.

In the design decisions of phase 1, we use the notion of a "logical access" to a record occurrence made by a transaction. This notion models the observation that in current database systems it is in general not possible to retrieve into main memory individual attributes, and, therefore, whenever a record occurrence is accessed, all attributes within it are accessed. After the fragments have been defined (and allocated), the physical decisions will be made so that the logical accesses of the most important transactions can be performed efficiently.

In the design decisions of phase 2, we use cost factors that do not depend on features of specific file or database system organizations. We realize that our model can be enhanced by using the specific information from a given DBMS, such as the available storage structures, file organizations, access methods,

transaction processing strategies, and so on. But such a specific model can only be used for a particular environment. We address the problem in a general way, so that our method yields a general, yet reasonable, solution.

Identification of Occurrences Within Fragments. In an environment where a single object is fragmented and allocated to one or more sites, it is necessary that every object occurrence (tuple) in fragments be fully identified. This can be done in two ways:

(i) by replicating the primary key of the object in every fragment;
(ii) by using tuple identifiers (TID), that is, system controlled identifiers for each tuple of the original object, which are replicated into all the fragments.

If solution (i) is used, the attributes constituting the primary key are not considered by the following optimization procedure, and are simply prepended to each fragment after the optimization; if solution (ii) is used, then the attributes constituting the primary key are treated like any other attributes.

1.2 Previous Related Work

The previous work on vertical partitioning can be grouped into two major categories. In the first, an optimal solution of the problem is attempted, typically under restrictive assumptions; in the second, a heuristic approach is used. We are not aware of any systematic work on the vertical partitioning with overlapping fragments and of its application in the context of distributed databases.

Hoffer [12] developed a nonlinear, zero–one program for the solution of the vertical partitioning problem which minimizes a linear combination of storage, retrieval, and update costs, under capacity constraints assigned to each subfile. Babad [1] formulated the less restricted vertical partitioning problem for variable length attributes as a nonlinear, zero–one program. In his model, one subrecord is designated as the primary record, and all requests are addressed to that subrecord and "dispatched" to other subrecords later. It is revealing that the final example considered four attributes only, as the problem soon becomes unmanageable.

The simpler problem of partitioning a record between a fast, primary memory and a slow, secondary memory has been formulated by Eisner and Severance in [9]. The authors show the isomorphism of this problem with the mincut–max flow network problem, which can be solved by the standard Ford/Fulkerson algorithm. A disadvantage of the method is that the graph requires a number of nodes and edges proportional to the sum of all attributes, users, and individual attributes required by each user; this number can become very large in real-life problems. In a follow-up paper, March and Severance [16] extended the previous model to incorporate block factors for both primary and secondary memories.

Schkolnick [20] considered the problem of clustering records within an IMS-type hierarchical structure, and determined an efficient solution method whose complexity is linear in the number of nodes of the hierarchical tree.

Papers of the second category describe heuristic approaches to the attribute partitioning problem. We agree with Hammer and Niamir [11] that the general vertical partitioning problem is "heuristic in nature," and that optimal methods cannot deal with it adequately.

Hoffer and Severance [13] measured the "affinity between pairs of attributes," and used the bond energy algorithm (BEA), developed in [17], to cluster attributes according to their affinity. Their attribute similarity function uses the notion of an attribute being "required for processing," or "required only if the record is selected," or finally "not required." We feel that in database systems the first and the second cases cannot be distinguished because all attributes of the same record are accessed together; therefore, we prefer the notion of logical accesses. We do, however, distinguish between retrieval and update accesses because, when an attribute is replicated in different fragments, an update transaction involving that attribute has to visit all fragments. In [13] attribute length is used in the evaluation of affinity—on the grounds that it should be less appealing to store attributes together as their length increases and more appealing as their length decreases; we do not consider length as a factor in determining affinity.

Hammer and Niamir [11] summarize the problems left unsolved by the approach in [12] in the following two points:

(i) The BEA determines an ordering of attributes, but it is still left to the subjective judgment of the designer to decide how to "clump" the attributes together to form fragments.
(ii) Similarity of pairs of attributes can be inadequate if the similarity among larger groups of attributes is not taken into account.

Our present paper extends the results of [13] by giving algorithms for the design of fragments that perform the activity at point (i) automatically and by taking into account in our algorithms "blocks" of attributes with similar properties, addressing point (ii).

Hammer and Niamir [11] developed two heuristics to be used in combination. The "grouping" heuristic starts by assigning each attribute to a different partition; at each step, all groupings of partitions of the current candidate solution are considered, and the one that represents the greatest improvement over the current candidate grouping becomes the new candidate. The process iterates until no improvement is found over the current candidate. The "regrouping" heuristic attempts to move attributes from their current partition to another partition. Grouping and regrouping are iterated until no further benefit can be achieved. When the set of all partitions is viewed as a lattice, grouping moves attributes "up" and regrouping moves attributes "sideways" [11].

The major criticism of this approach is regarding the direction in which to explore the lattice—while Hammer and Niamir proceed "up" by grouping, we proceed "down" by splitting. The rationale behind this approach is that the "optimal" solution, in our opinion, is much closer to the group composed of all attributes, assumed to be our starting point, than to groups that are single-attribute partitions, as assumed in [11]. A typical solution will not incur too much splitting because of the disadvantage of storing too many small subrecords. Thus, in our approach, the "moves" along the lattice will be in a direction that promises optimality; Hammer and Niamir, on the contrary, recognized that their approach introduces many discrepancies at the first iterations (in fact, they introduced the regrouping heuristics to get rid of these discrepancies).

Some other research is related to the present paper. Chang and Cheng discussed both horizontal and vertical partitioning [6]. Their paper uses the term *horizontal*

partitions for the issues addressed here (usually called *vertical partitioning*). They developed a methodology for the decomposition of an object into fragments, but did not give a precise evaluation of the factors that should guide the decomposition. Dayal and Bernstein [8] have studied the fragmentation problem from a more theoretical viewpoint, giving rules through which objects can be lossless decomposed into fragments.

Ceri, Negri, and Pelagatti [3] analyzed the horizontal partitioning problem, dealing with the specification of partitioning predicates and the application of horizontal partitioning to different database design problems. Ceri, Navathe, and Wiederhold [4] have proposed a model for designing distributed database schemas that incorporates the notion of horizontal partitioning. The paper describes a linear integer formulation of the horizontal partitioning problem without replication and then presents heuristic algorithms for the decomposition of the problem into smaller subproblems and the introduction of replication, assuming the nonreplicated solution as a starting point. The present paper complements our past work on horizontal partitioning.

Finally, several papers have addressed the general problem of allocating database objects over multiple sites, without taking object partitioning into consideration; among them are [7, 10, 15, 18].

1.3 Algorithms for the Vertical Partitioning Problem—Organization of the Paper

In this section we provide an overview of the structure of the paper by discussing the various algorithms that are used to solve the VPP. The algorithms are discussed in detail in Sections 2 and 3.

As shown in Figure 2, the input to the algorithms is twofold:

(a) The *logical accesses* of transactions to the object.
(b) The *relevant design parameters*, such as cost of storage, cost of accessing a record occurrence, cost of transmission.

Input (a) is required for phase 1 of the VPP; input (b) for phase 2.

The given information about the usage of attributes by transactions is initially converted into a square matrix, called the *attribute affinity matrix*. This matrix is diagonalized by the algorithm CLUSTER, described in Section 2.1, which is essentially the bond energy algorithm [17]. In that paper the algorithm has been shown to be useful in partitioning a set of interacting variables into subsets such that the subsets either do not interact or interact minimally.

We distinguish the following three environments for the vertical partitioning problem:

(a) single site with one memory level;
(b) single site with several memory levels (i.e., devices of different speeds, capacities, and costs); and
(c) multiple sites.

Problem (a) can be solved by using either nonoverlapping or overlapping fragments. These are produced by the use of the algorithms SPLIT_NONOVERLAP and SPLIT_OVERLAP, respectively—described in Sections 2.2 and 2.3; these two algorithms automate the manual grouping of attributes required in [13]. The algorithms are invoked repeatedly (see Section 2.6) until no further partition

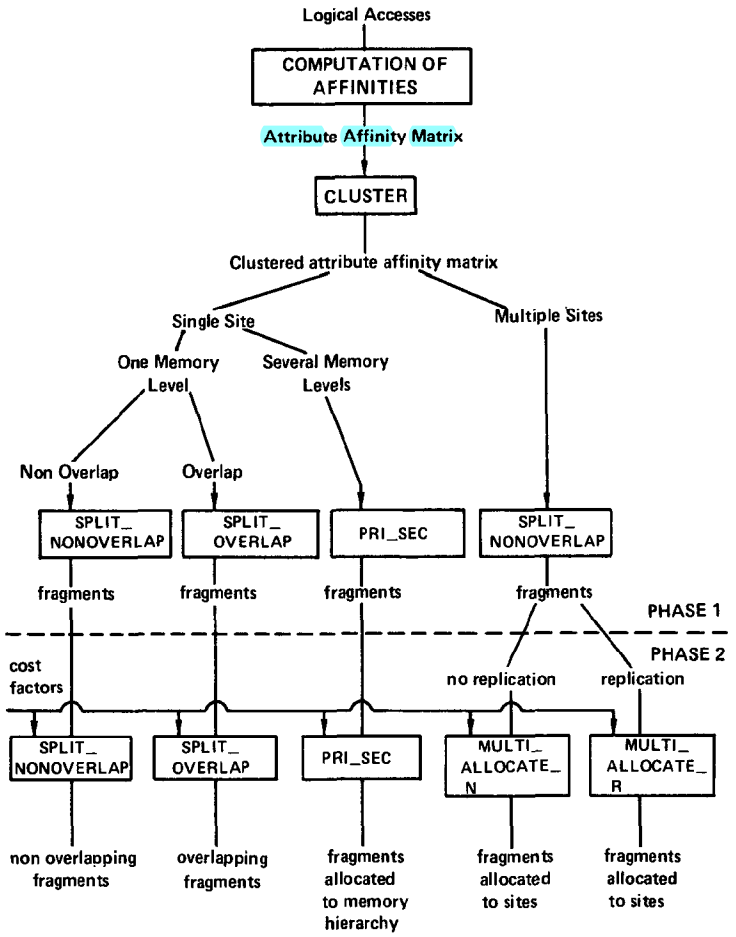

Fig. 2. Algorithms for solving the vertical partitioning problem in different application environments.

occurs; they can use empirical objective functions for phase 1, described in Sections 2.2 and 2.3, or objective functions based on cost factors for phase 2, described in Section 3.2.

Problem (b) is a special application of the VPP, where an object may be resident on secondary memory and only a "highly active" part of it will be stored in primary (fast) memory. This problem uses the PRI_SEC algorithm, described in Section 2.4, in place of the SPLIT algorithms mentioned above. The PRI_SEC algorithm can also use either an empirical objective function for phase 1, described in Section 2.4, or an objective function based on cost factors, described in Section 3.3. If the fragments have to be allocated among multiple levels of memory, the PRI_SEC algorithm needs to be applied at each level (see Section 3.3).

Problem (c) considers the VPP problem in a distributed database environment. The SPLIT_NONOVERLAP algorithm is used to design the fragments in phase 1; the allocation of fragments to sites requires cost factors and must be done in phase 2. Algorithms MUTLI_ALLOCATE_N and MULTI_ALLOCATE_R, de-

scribed in Sections 3.4 and 3.5, respectively, determine the allocation of fragments to sites with and without replication.

Variations of the algorithm are described throughout Section 2; in particular, the algorithm SHIFT, which produces the pairwise permutation of the first row and column of the clustered affinity matrix with the last rows and columns, and the algorithm CLUSTER, which recomputes affinity after producing intermediate partitionings, can be used at every iteration of the SPLIT algorithms. We anticipate that these algorithms will be used in a highly interactive setting, where the designer can select the best combination of algorithms. Conclusions and further research problems are described in Section 4.

2. ALGORITHMS BASED ON EMPIRICAL OBJECTIVE FUNCTIONS

Vertical partitioning algorithms require the specification of data and transaction parameters shown in Tables I and II.

Table I presents the parameters that refer to a database object having ten attributes on a single site. The 0/1 entries in the ATTRIBUTE USAGE (AU) matrix show whether or not a given attribute is used by a given transaction. The ACCESS column shows for each transaction the frequency of access to object instances per unit time period (e.g., a day). The ATTRIBUTE LENGTH (AL) row gives the number of bytes of each attribute. The TYPE column shows the type of the transaction, depending upon whether the transaction retrieves (R) or updates (U) it. In this paper, for ease of notation, we give the same treatment to attributes within one transaction (i.e., all attributes are considered to be either retrieved or updated). In an actual environment some attributes may be retrieved and some may be updated; this calls for recording some additional information for transactions that involve updates. Dealing with this general case, however, does not change the basic structures of the algorithms.

Table II shows the parameters for a multiple-site problem with four sites. The access column is now transformed into an ACCESS MATRIX, giving the number of transaction accesses from each site.

In the single object (file) VPP problem all the attributes of the object appear in the attribute usage matrix and attribute length row; in a database VPP problem, one can either imagine one data usage per object or a global data usage table of all the transactions against all the attributes. From a global table, one would select the relevant columns for a given object and all transactions that utilize at least one attribute from the object, and construct individual data usage tables.

2.1 Affinity Among Attributes

The first step in the design of a vertical partition is to construct an ATTRIBUTE AFFINITY (AA) matrix. Figure 1 shows an example of an attribute affinity matrix derived from the attribute usage in Table I. AA is a symmetric square matrix which records the affinity among the attributes $a_{i'}$ and $a_{i''}$ as a single number, $\text{aff}_{i'i''}(= \text{aff}_{i''i'})$, defined below.

Let the following parameters be defined for each transaction k:

$u_{ki} = 1$ if transaction k uses attribute a_i
$= 0$ otherwise (u_{ki} is an element of matrix AU).

Table I. Parameters for a Single Site VPP

	Attribute usage matrix										Type	Single site accesses per time period
Attribute	1	2	3	4	5	6	7	8	9	10		
T 1	1	0	0	0	1	0	1	0	0	0	Type 1 = R	Acc 1 = 25
T 2	0	1	1	0	0	0	0	1	1	0	Type 2 = R	Acc 2 = 50
T 3	0	0	0	1	0	1	0	0	0	1	Type 3 = R	Acc 3 = 25
T 4	0	1	0	0	0	0	1	1	0	0	Type 4 = R	Acc 4 = 35
T 5	1	1	1	0	1	0	1	1	1	0	Type 5 = U	Acc 5 = 25
T 6	1	0	0	0	1	0	0	0	0	0	Type 6 = U	Acc 6 = 25
T 7	0	0	1	0	0	0	0	0	1	0	Type 7 = U	Acc 7 = 25
T 8	0	0	1	1	0	1	0	0	1	1	Type 8 = U	Acc 8 = 15
	Attribute length vector											
	10	8	4	6	15	14	3	5	9	12		

Table II. Additional Parameters for a Multiple Site VPP

	Multiple site accesses per time period			
Site	1	2	3	4
T 1	10	15	0	0
T 2	10	20	10	10
T 3	0	0	15	10
T 4	10	15	0	10
T 5	5	10	5	5
T 6	10	5	5	5
T 7	5	10	5	5
T 8	5	5	2	3

type_k = 'r' (retrieval transactions)
= 'u' (update transactions).
n_k = number of accesses to object instances for one occurrence of transaction k (single site).
n_{kj} = number of accesses to object instances for one occurrence of transaction k at site j (multiple sites).
freq_k = frequency of occurrence of transaction k (single site).
freq_{kj} = frequency of occurrence of transaction k at site j (multiple sites).
$\text{acc}_k = n_k \text{freq}_k$ (single site)
$= \sum_j n_j \text{freq}_{kj}$ (multiple sites).

acc_k measures the number of accesses of transaction k to the object instances per time period.

The affinity measure is

$$\text{aff}_{i'i''} = \sum_{k \mid u_{ki'}=1 \wedge u_{ki''}=1} \text{acc}_k.$$

The attribute affinity defined above measures the strength of an imaginary bond between the two attributes, on the basis of the fact that attributes are used together by transactions. Note that affinity as defined here is *not* site specific; we use affinity information for determining the composition of fragments. In Section 3 we shall discuss how the site-specific access measures can be employed to determine the allocation of fragments in distributed databases.

The algorithms developed below will cluster groups of attributes with high affinity in the same fragment, and keep attributes with low affinity in separate fragments. One may refine the notion of affinity to take different factors into account; the elaborate formulation of affinity given in [13] could be used. We stress that the particular definition of affinity has no bearing whatsoever on the design of the subsequent algorithms; all they presuppose is that the affinity matrix has been defined.

2.2 Clustering of Attributes

The AA matrix is processed by the algorithm CLUSTER. The algorithm employs the bond energy algorithm (BEA) of [17], which is a general procedure for

permuting rows and columns of a square matrix in order to obtain a semiblock diagonal form. The algorithm is typically applied to partition a set of interacting variables into subsets which interact minimally.

The objective function used by the BEA tends to surround large values of the matrix with large values, and the small ones with small values. It maximizes the following expression:

$$\sum_{ij} \text{aff}_{ij}(\text{aff}_{i,j-1} + \text{aff}_{i,j+1} + \text{aff}_{i-1,j} + \text{aff}_{i+1,j})$$

where

$$\text{aff}_{i,0} = \text{aff}_{0,j} = \text{aff}_{i,n+1} = \text{aff}_{n+1,j} = 0.$$

The algorithm is heuristic, and it considers only a small number of the possible permutations of the columns; it has a complexity $O(n^2)$. The algorithm takes advantage of the following features:

(i) the AA matrix is symmetric, and hence allows a pairwise permutation of rows and columns, which reduces the complexity;

(ii) because of the definition of $\text{aff}_{i'i''}$, the initial AA matrix is already semiblock diagonal, in that each diagonal element has a greater value of any element along the same row or column ($\text{aff}_{i'i'} \geq \text{aff}_{i'i''}$ and $\text{aff}_{i''i''} \geq \text{aff}_{i''i'}$, for any i' and i''). This feature motivates, from a practical point of view, the definition of diagonal elements of the matrix, which have no physical meaning.

The application of the CLUSTER algorithm to the initial AA matrix of Figure 3a produces the new AA matrix in Figure 3b. The blocks along the main diagonal of AA correspond to groups of jointly accessed data items which are candidates for constituting vertical fragments.

So far our procedure is similar to that of [13]. After diagonalizing the AA matrix, Hoffer and Severance suggest a tentative assignment of groups of attributes to fragments. We have designed the three basic algorithms: SPLIT_NON_OVERLAP, SPLIT_OVERLAP, and PRI_SEC in order to perform a less subjective clustering of attributes into fragments for the single-site, binary partitioning problems;[1] extensions of the basic algorithms allow one to deal with n-ary partitions and multiple sites.

2.3 Partitioning an Object Into Nonoverlapping Fragments

The SPLIT_NON_OVERLAP algorithm uses the clustered affinity matrix to partition an object into two nonoverlapping fragments. Assume that a point x is fixed along the main diagonal of the clustered AA matrix, as shown in Figure 3c. The point x defines two blocks U (for "upper") and L (for "lower"). Each block defines a vertical fragment given by the set of attributes in that block.

Starting from the attribute usage matrix, we define $A(k)$ as the set of attributes used by transaction k as follows:

$$A(k) = \{i \mid u_{ki} = 1\}.$$

[1] These algorithms were proposed by S. Ceri and implemented by J. Dou and S. Ceri.

Attribute	1	2	3	4	5	6	7	8	9	10
1	75	25	25	0	75	0	50	25	25	0
2	25	110	75	0	25	0	60	110	75	0
3	25	75	115	15	25	15	25	75	115	15
4	0	0	15	40	0	40	0	0	15	40
5	75	25	25	0	75	0	50	25	25	0
6	0	0	15	40	0	40	0	0	15	40
7	50	60	25	0	50	0	85	60	25	0
8	25	110	75	0	25	0	60	110	75	0
9	25	75	115	15	25	15	25	75	115	15
10	0	0	15	40	0	40	0	0	15	40

(a)

Attribute	5	1	7	2	8	3	9	10	4	6
5	75	75	50	25	25	25	25	0	0	0
1	75	75	50	25	25	25	25	0	0	0
7	50	50	85	60	60	25	25	0	0	0
2	25	25	60	110	110	75	75	0	0	0
8	25	25	60	110	110	75	75	0	0	0
3	25	25	25	75	75	115	115	15	15	15
9	25	25	25	75	75	115	115	15	15	15
10	0	0	0	0	0	15	15	40	40	40
4	0	0	0	0	0	15	15	40	40	40
6	0	0	0	0	0	15	15	40	40	40

(b)

Attribute	5	1	7	2	8	3	9	10	4	6
5	75	75	50	25	25	25	25	0	0	0
1	75	75	50	25	25	25	25	0	0	0
7	50	50	85	60	60	25	25	0	0	0
2	25	25	60	110	110	75	75	0	0	0
8	25	25	60	110	110	75	75	0	0	0
3	25	25	25	75	75	115	115	15	15	15
9	25	25	25	75	75	115	115	15	15	15
10	0	0	0	0	0	15	15	40	40	40
4	0	0	0	0	0	15	15	40	40	40
6	0	0	0	0	0	15	15	40	40	40

(c)

Fig. 3. (a) Initial attribute affinity matrix (AA); (b) attribute affinity matrix in semiblock diagonal form; (c) nonoverlapping splitting of AA into two blocks L and U.

Using $A(k)$, it is possible to compute the following sets:

$$T = \{k \mid k \text{ is a transaction}\}$$

$$LT = \{k \mid A(k) \subseteq L\}$$

$$UT = \{k \mid A(k) \subseteq U\}$$

$$IT = T - \{LT \cup UT\}$$

T represents the set of all transactions; LT and UT represent the set of transactions that "match" the partitioning, as they can be entirely processed using attributes in the lower or upper block, respectively; IT represents the set of transactions that needs to access both fragments.

$$CT = \sum_{k \in T} \text{acc}_k$$

$$CL = \sum_{k \in LT} \text{acc}_k$$

$$CU = \sum_{k \in UT} \text{acc}_k$$

$$CI = \sum_{k \in IT} \text{acc}_k$$

CT counts the total number of transaction accesses to the considered object. CL and CU count the total number of accesses of transactions that need only one fragment; CI counts the total number of accesses of transactions that need both fragments. We consider $n - 1$ possible locations of point x along the diagonal, where n is the size of the matrix (i.e., the number of attributes). A nonoverlapping partition is obtained by selecting the point x such that the following goal function z is maximized:

$$\max z = CL \times CU - CI^2. \tag{1}$$

The partitioning that corresponds to the maximal value of the z function is accepted if z is positive, and is rejected otherwise.

The above objective function comes from an empirical judgment of what should be considered a "good" partitioning. The function is increasing in CL and CU and decreasing in CI. For a given value of CI, it selects CL and CU in such a way that the product $CL \times CU$ is maximized. This results in selecting values for CL and CU that are as nearly equal as possible. Thus, the above z function will produce fragments that are "balanced" with respect to the transaction load. Notice that in Section 2.5, where we deal with memory hierarchies, we present a goal function that attains the opposite result (i.e., produces "unbalanced" partitionings).

Recalling that $CT = CL + CU + CI$ is a constant value, it is possible to visualize the z surface in the space of CL and CU (see Figure 4). Notice that $z = 0$ in absence of a partitioning, where either ($CL = 0$ and $CU = CT$) or ($CU = 0$ and $CL = CT$) and $CI = 0$; z takes its minimum feasible value at (0, 0), where it is $z = -CT^2$, and its maximum feasible value at ($CT/2$, $CT/2$), where it is $z = CT^2/4$. Figure 4 also shows the intersection of z with the $z = 0$ plane; partitions corresponding to points in the dashed area are accepted (because it is $z \geq 0$).

The proposed algorithm has the disadvantage of not being able to partition an object by selecting out an embedded "inner" block. For example, consider the clustering of attributes shown in the AA matrix in Figure 5a, which is a reasonable result for the bond energy algorithm (the example was actually produced by applying the BEA algorithm). The best binary partition in this case is the one that places the attributes between points $x1$ and $x2$ on the diagonal into one fragment, and all other attributes into the other; this solution is not one of those considered by the SPLIT_NON_OVERLAP algorithm.

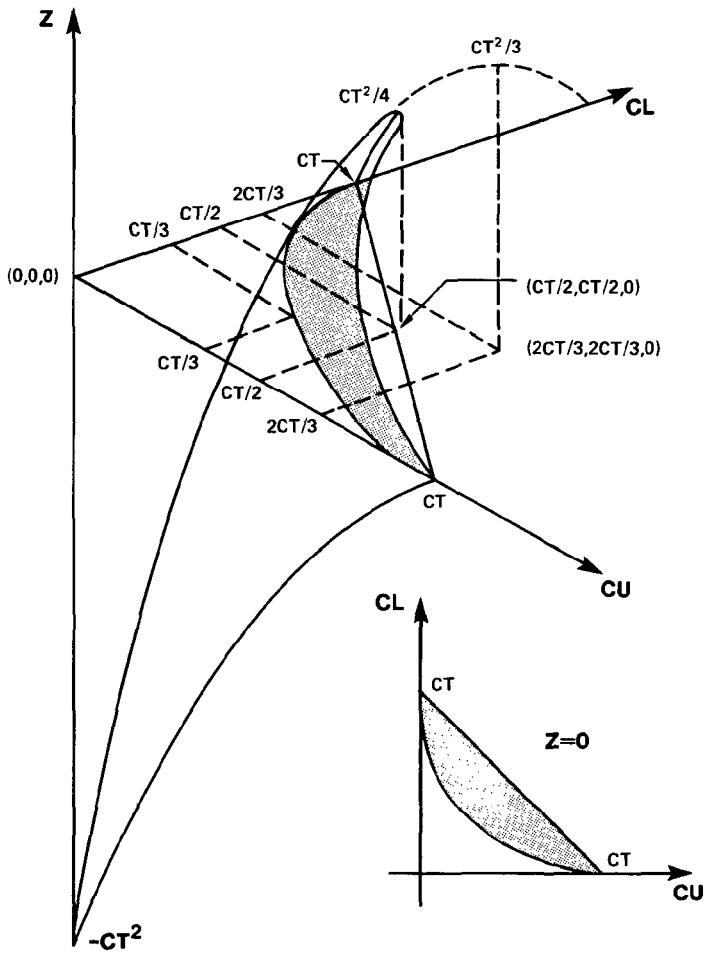

Fig. 4. A surface representing the function z used by the SPLIT_NON_OVERLAP algorithm.

This disadvantage can be avoided by using the procedure SHIFT, which moves the leftmost column of the AA matrix to the extreme right, and the topmost row of the matrix to the bottom (see Figure 5b). SHIFT is called a total of n times, so that every diagonal block gets the opportunity of being brought to the upper left corner in the matrix; the SPLIT_NON_OVERLAP algorithm is used to select the best partition after each application of SHIFT. Figure 5c shows the AA matrix in which, by applying SHIFT three times, the "inside" block of Figure 5a has been taken to the upper left corner. When the SHIFT procedure is used, the complexity of the algorithm increases by factor n. Experience has shown that the use of the SHIFT procedure improves the solution of the BVP problem in several cases.

2.4 Partitioning an Object Into Overlapping Fragments

A binary overlapping splitting consists of two clusters of attributes with a nonempty intersection. A representation of overlapping clusters on the AA matrix

Attribute	3	2	1	7	6	4	5	10	8	9
3	10	10	10	0	0	0	0	0	0	0
2	10	10	10	0	0	0	0	0	0	0
1	10	10	10	0	0	0	0	0	0	0
			x1							
7	0	0	0	50	50	50	50	0	0	0
6	0	0	0	50	50	50	50	0	0	0
4	0	0	0	50	50	50	50	0	0	0
5	0	0	0	50	50	50	50	0	0	0
								x2		
10	0	0	0	0	0	0	0	10	10	10
8	0	0	0	0	0	0	0	10	10	10
9	0	0	0	0	0	0	0	10	10	10

(a)

Attribute	2	1	7	6	4	5	10	8	9	3
2	10	10	0	0	0	0	0	0	0	10
1	10	10	0	0	0	0	0	0	0	10
7	0	0	50	50	50	50	0	0	0	0
6	0	0	50	50	50	50	0	0	0	0
4	0	0	50	50	50	50	0	0	0	0
5	0	0	50	50	50	50	0	0	0	0
10	0	0	0	0	0	0	10	10	10	0
8	0	0	0	0	0	0	10	10	10	0
9	0	0	0	0	0	0	10	10	10	0
3	10	10	0	0	0	0	0	0	0	10

(b)

Attribute	7	6	4	5	10	8	9	3	2	1
7	50	50	50	50	0	0	0	0	0	0
6	50	50	50	50	0	0	0	0	0	0
4	50	50	50	50	0	0	0	0	0	0
5	50	50	50	50	0	0	0	0	0	0
10	0	0	0	0	10	10	10	0	0	0
8	0	0	0	0	10	10	10	0	0	0
9	0	0	0	0	10	10	10	0	0	0
3	0	0	0	0	0	0	0	10	10	10
2	0	0	0	0	0	0	0	10	10	10
1	0	0	0	0	0	0	0	10	10	10

(c)

Fig. 5. (a) Clustering of attributes with a block "inside" the matrix; (b) the SHIFT procedure after one iteration; (c) the best nonoverlapping split at the third iteration of the SHIFT procedure.

is given in Figure 6: notice that now two points x_1 and x_2 are needed along the diagonal. The upper fragment comprises the attributes from the first one up to x_2, and the lower comprises the attributes from x_1 to the last one; attributes between x_1 and x_2 constitute the intersection.

Attribute	5	1	7	2	8	3	9	10	4	6
5	75	75	50	25	25	25	25	0	0	0
1	75	75	50	25	25	25	25	0	0	0
7	50	50	85	60	60	25	25	0	0	0
2	25	25	60	110	110	75	75	0	0	0
8	25	25	60	110	110	75	75	0	0	0
						x1				
3	25	25	25	75	75	115	115	15	15	15
9	25	25	25	75	75	115	115	15	15	15
							x2			
10	0	0	0	0	0	15	15	40	40	40
4	0	0	0	0	0	15	15	40	40	40
6	0	0	0	0	0	15	15	40	40	40

Fig. 6. Overlapping splitting into two blocks L and U.

In the algorithm SPLIT_OVERLAP we consider the $n \times (n-1)/2$ possible combinations of locating points x_1 and x_2 along the main diagonal. The case with $x_1 = x_2$, corresponding to nonoverlapping splittings, is also considered.

The same objective function is used as in Eq. (1) above, but with a different evaluation of sets LT and UT. Transactions that update attributes from the intersection $I = L \cap U$ have to be directed to both fragments in order to preserve the consistency of replicated data. On the other hand, by sharing the attributes from I in both L and U, more read transactions can be satisfied by either L or U alone, as compared with the nonoverlapping case. As before, LT and UT represent the set of transactions that "match" the two fragments, while IT represents the set of transactions that need to access both fragments. We have

$$LT = \{k \mid (\text{type}_k = \text{'r'} \wedge A(k) \subseteq L) \vee (\text{type}_k = \text{'u'} \wedge A(k) \subseteq (L - I))\}$$

$$UT = \{k \mid (\text{type}_k = \text{'r'} \wedge A(k) \subseteq U) \vee (\text{type}_k = \text{'u'} \wedge A(k) \subseteq (U - I))\}$$

$$IT = T - (LT \cup UT).$$

From the above sets, we compute the counts CL, CU, CI as before and use them in the goal function (1). To keep the formulation free of cost factors in the empirical design phase, we do not incorporate the additional storage cost of attributes in I here. This cost factor is instead taken into account in the cost optimization phase of Section 3.

Notice that $LT \cap UT$ is not void (it includes the retrieval transactions that use attributes from the intersection I only (i.e., $(A(k) \subseteq I)$), and therefore the equation $CT = CL + CU + CI$ does not hold any more.

It is also possible to use the SHIFT procedure described in the previous section in the SPLIT_OVERLAP algorithm. As before, the complexity of the problem increases by factor n.

2.5 Partitioning Into Primary and Secondary Fragments

The PRI_SEC algorithm is designed to produce a "biased" partitioning into a primary and a secondary fragment. It is applicable to the problem of allocating an object to multiple-storage devices.

Attribute	5	1	7	2	8	3	9	10	4	6
5	75	75	50	25	25	25	25	0	0	0
1	75	75	50	25	25	25	25	0	0	0
7	50	50	85	60	60	25	25	0	0	0
2	25	25	60	110	110	75	75	0	0	0
8	25	25	60	110	110	75	75	0	0	0
3	25	25	25	75	75	115	115	15	15	15
9	25	25	25	75	75	115	115	15	15	15
10	0	0	0	0	0	15	15	40	40	40
4	0	0	0	0	0	15	15	40	40	40
6	0	0	0	0	0	15	15	40	40	40

Fig. 7. Splitting into a primary (U) and a secondary (L) fragment.

Consider a BVP where a logical object is to be partitioned into two blocks U and L; U is the block containing the "more important" attributes (i.e., those used more frequently) and L is the block containing the "less important" ones. Attributes in U will typically be assigned to the faster (PRImary) memory, while those in L (or the entire object) to a slower (SECondary) memory.

The PRI_SEC algorithm partitions the clustered AA matrix into two nonoverlapping blocks L and U that meet the above criteria. To do so, it is necessary to bring the "less important" block to the bottom right corner of the matrix (see Figure 7). This is accomplished by the procedure SHIFT, described in Section 2.3, called by PRI_SEC. Also in this case, SHIFT is called a total of n times, so that every block gets the opportunity of being brought to the lower right corner in the matrix. After each shift of the AA matrix, every point x along the main diagonal is considered for defininining L and U; thus, $(n-1) \times n$ possible solutions are examined.

Let the following parameters and counts be defined:

$$l_i\text{: length of attribute } i,$$

$$TL = \sum_i l_i\text{: total length of the object,}$$

$$UL = \sum_{i \in U} l_i\text{: length of attributes in the upper block.}$$

CT and CU are computed as in the nonoverlapping problem (see Section 2.3).

The partition evaluation function needs to be asymmetric in order to maximize the number of transactions that are satisfied by using just the attributes of U. By the very fact that we are considering this problem, we have assumed that the entire object cannot be conveniently stored in primary memory. Hence, a counteracting objective is to keep the size of U, in terms of the total length of attributes in it, to a minimum. This effect is produced by maximizing the following objective function:

$$\max z = \frac{CU}{CT} - \frac{UL}{TL}. \tag{2}$$

The splitting that corresponds to the maximal value of the z function is accepted if z is positive, and rejected otherwise.

In this function, CU/CT is the ratio of the transaction volume satisfied by the upper block (U) to the total transaction volume; UL/TL is the ratio of the size of the fragment defined by U to the size of the object. Whenever z is positive, the former ratio exceeds the latter, implying that the primary block (U) is able to carry a fraction of the transaction load that is higher in comparison to its relative size. Maximizing this function intuitively selects a "good" block U. The above objective function was selected in preference to several other candidates—one worth mentioning, which gives similar results, is

$$\max z = \frac{CU \times TL}{CT \times UL}. \tag{2'}$$

2.6 Repetitive Use of Binary Partitioning

The solution of the general VPP problem may consist of several fragments; thus the algorithms that we have presented for the BVP problem have to be extended from binary to n-ary partitioning. Consider the AA matrix in Figure 8; assume that the three well-defined blocks shown in the figure correspond to the final fragmentation. If a nonoverlapping n-ary partitioning is to be accomplished by an algorithm, the algorithm in general has to try using 1 or 2 or . . . $n - 1$ points along the diagonal as candidate break points to define the partitions. The complexity of such algorithm would be $O(2^n)$. The complexity of an algorithm for the overlapping problem would be even greater.

To avoid the combinatorial complexity, we propose solving the VPP problem by repetitively applying the BVP algorithms. At each iteration the fragments produced by the algorithms are considered as two distinct objects, and the BVP algorithm is applied to further partition each of them. This generates two distinct subproblems. In each subproblem only those transactions are considered by the SPLIT algorithms that have some intersection with the considered fragment, and all other transactions are disregarded; thus, decisions about further splitting are not influenced by irrelevant transactions.[2] The process terminates when no further partitioning is indicated.

This approach is suboptimal compared to the general n-ary partitioning, but it drastically reduces the complexity of the computation of the n-ary VPP; for large n, the overall complexity can be considered equivalent to the complexity of the first application of SPLIT. The approach is not only computationally very attractive, but it takes advantage of the following two facts:

(i) The selection of the best partition at each binary split corresponds to taking the most promising decision; to this extent, the approach can be classified as a "greedy" heuristic.

(ii) At each iteration of the solution procedure, new fragments are generated which give rise to a new BVP for each fragment. These subproblems are totally independent, in the sense that each considers a fragment as an object in its own right. Each new subproblem considers all the relevant

[2] A slight error is introduced in the overlapping case for retrieval transactions that use attributes from intersection I only (i.e., $A(k) \subseteq I$), because these transactions are considered in both problems, while a transaction will eventually use one of the fragments only; this error can be avoided by arbitrarily assigning these transactions to one of the two fragments.

Attribute	5	1	7	2	8	3	9	10	4	6
5	75	75	50	25	25	25	25	0	0	0
1	75	75	50	25	25	25	25	0	0	0
7	50	50	85	60	60	25	25	0	0	0
2	25	25	60	110	110	75	75	0	0	0
8	25	25	60	110	110	75	75	0	0	0
3	25	25	25	75	75	115	115	15	15	15
9	25	25	25	75	75	115	115	15	15	15
10	0	0	0	0	0	15	15	40	40	40
4	0	0	0	0	0	15	15	40	40	40
6	0	0	0	0	0	15	15	40	40	40

Fig. 8. A n-ary partitioning.

Attribute Part*	5	1	7	2	8	3	9	10	4	6	Iteration level
1	X	X	X								2
2				X	X	X	X				2
3								X	X	X	1

Attribute Part*	5	1	7	2	8	3	9	10	4	6	Iteration level
1	X	X	X								2
2			X	X	X						2
3			X	X	X						2
4				X	X	X	X				3
5								X	X	X	3

Attribute	5	1	7	2	8	3	9	10	4	6	Iteration level
PRI	X	X	X	X	X	X	X				1
SEC								X	X	X	
PRI	X	X	X	X	X						2
SEC						X	X	X	X	X	
PRI	X	X	X	X							3
SEC					X	X	X	X	X	X	

* abbreviates partition in all subsequent figures.

Fig. 9. Results of repetitive application of SPLIT_NON_OVERLAP, SPLIT_OVERLAP, and PRI_SEC.

transactions that use attributes of the fragment, without any loss of information from the original problem.

Figure 9 shows the output of the repeated application of the algorithms SPLIT_NON_OVERLAP, SPLIT_OVERLAP, and PRI_SEC to the partitioning problem whose input parameters are shown in Table I.

In Figures 9a and 9b, the output includes the final fragmentation produced by the SPLIT algorithm. Each fragment in the final solution is associated with the iteration level in the binary split tree at which that fragment was produced. For example, in Figure 9a, the fragment including attributes 10, 4, and 6 was produced at the first iteration, and the remaining fragment including attributes 5, 1, 7, 2, 8, 3, and 9 was further divided into two fragments, including attributes 5, 1, 7 and 2, 8, 3, 9, respectively, at the second iteration.

In Figure 9c, the output includes the solutions produced at each iteration of the BVP problem using the PRI_SEC algorithm; the final solution is to have attributes 5, 1, 7, and 2 in the primary memory.

In the nonoverlapping and overlapping algorithms, the iterative approach can take advantage of the following two options in the BVP algorithms:

(i) the evaluation of the affinity measures can be repeated for each subproblem independently, by considering only the relevant transactions. Procedure CLUSTER is then applied to produce the semiblock diagonal form of the new affinity matrix.
(ii) the procedure SHIFT can be called by the SPLIT algorithms; notice that the use of the procedure SHIFT is mandatory in the case of the PRI_SEC algorithm.

Computational experience has shown that the repetitive application of CLUSTER improves the performance of the partitioning algorithms, because decisions that are made at each iteration rely on current values of attribute affinity. On the other hand, the application of SHIFT does not substantially improve the performance of partitioning algorithms. Recall the discussion in Section 2.3 about the need of selecting blocks "inside" the affinity matrix for the BVP. In the VPP, without applying SHIFT, it is likely that "inside" blocks are selected after some iterations, thus the two versions of SPLIT algorithms, with and without SHIFT, lead to the same result with a different sequence of splittings. Notice that the use of SHIFT at the first iteration is very expensive, since it increases the overall complexity of the VPP problem by factor n.

It is possible to enforce some constraints on the partitioning process:

(i) Impose a limitation on the number of partitions that can be produced; this constraint is motivated by the limitation on the overall number of files that can be supported by the system.
(ii) Impose a minimum size on fragments; this constraint can prevent partitioned records from becoming too small.

These constraints and the appropriate use of CLUSTER and SHIFT procedures can be properly controlled by a knowledgeable designer. We contemplate the use of an interactive design tool in which all these procedures are available to the designer, intermediate results are shown, and the designer is able to direct the design sequence.

3. VERTICAL PARTITIONING BASED ON COST OPTIMIZATION

In the previous section we have given algorithms that use empirical objective functions, suggested by our intuition of the problem. The disadvantage of the approach is in its lack of precision; such an approach is attractive, however, as it

Problem type	Cost factors			
	C1	C2	C3	C4
S	M	H	L	—
H	L	H	H	—
M	L	M	M	H

Fig. 10. The relative importance (L = low, M = medium, H = high) of coefficients C1, C2, C3, and C4 in different application environments (S = single site; H = single site, several memory levels; M = multiple sites).

is possible to evaluate the convenience of partitioning without requiring absolute measurements of costs, which are seldom available or accurate.

As we suggested in Figure 1, the empirical design should be followed by a cost optimization phase. For each of the three application environments of VPP defined in Section 1.3, it is possible to estimate the total cost of vertical partitioning in terms of certain cost factors. They are

C1. cost of irrelevant attributes accessed within a fragment (per byte);
C2. cost of accessing fragments for retrieval and updates (per access);
C3. storage cost (per byte);
C4. transmission cost (per byte).

Note that the above cost factors are relative and that they represent an estimate of the design parameters associated with a particular physical environment. Figure 10 shows the relative importance (low, medium, or high) of these cost factors in the three types of application environments, as expressed in the database design literature in general [23] and as it applies to present-day systems. The relative importance of the factors stems from the following:

(a) *Single site with one memory level.* The costs C1 and C2 are both related to unnecessary I/O. The cost C2 also includes the CPU cost of putting the data from the fragments together and of applying updates simultaneously to fragments. Hence, C2 is the most important cost factor, followed by C1. In comparison, C3 is less important because storage is becoming less and less expensive, and C4 is not applicable.

(b) *Single site with several memory levels.* In this case, the costs of accessing fragments and of storage are both equally important, since there is a trade-off between them. C1 is relatively less important, and C4 is not applicable.

(c) *Multiple sites.* In the case of a distributed database, the transmission cost is dominant [2, 5]; however, local access and storage costs cannot be disregarded [21]. Therefore, we have shown C4 as having high importance, C2 and C3 as having medium importance, and C1 as having low importance.

In the following, we give simple expressions for weighing these cost factors in the three types of problems. The objective function we will consider throughout the rest of the paper is

$$\min z = \sum_{1 \le i \le 4} w_i c_i. \tag{3}$$

The VPP and solution algorithms for a specific problem are identified by a five-character subscript: $i1$, $i2$, $i3$, $i4$, $i5$

$i1$:	S	= single site, one memory level
	H	= single site, several memory levels (heterogeneous devices)
	M	= multiple sites
$i2$:	N	= multiple sites, nonreplicated
	R	= multiple sites, replicated
	—	= single site
$i3$:	N	= nonoverlapping
	O	= overlapping
	—	= several memory levels
$i4$:	S	= algorithm SHIFT used
	—	= algorithm SHIFT not used
$i5$:	C	= algorithm CLUSTER used
	—	= algorithm CLUSTER not used

Fig. 11. Classification of the VPP and of options within algorithms.

This objective function can be used in the SPLIT_NONOVERLAP, SPLIT_OVERLAP, and PRI_SEC algorithms instead of the empirical objective functions (1) and (2) described in Section 2.

3.1 Notation and Definitions

This subsection is devoted to the definition of some additional notation. It includes a summary of some notation from Section 2, which is retained. To denote one particular problem and solution method, we use five "switches," explained in Figure 11. Subscripts $i1$, $i2$, $i3$ define the nature of the problem, while subscripts $i4$, $i5$ define the choice of algorithms used for the solution. In Figures 12, 13, 15, and 16 the "problem type" column on the left contains this five-character subscript.

In a BVP problem, the sets LT and UT include the transactions that can be entirely processed in the upper (U) or lower (L) block; their definition is the same as in Section 2.4 (overlapping fragments), and it applies also to the nonoverlapping and the primary–secondary fragments, where the intersection $I = L \cap U$ is void. The set IT includes transactions that need to access both fragments.

$$LT = \{k \mid (\text{type}_k = \text{'r'} \wedge A(k) \subseteq L) \vee (\text{type}_k = \text{'u'} \wedge A(k) \subseteq (L - I))\}$$

$$UT = \{k \mid (\text{type}_k = \text{'r'} \wedge A(k) \subseteq U) \vee (\text{type}_k = \text{'u'} \wedge A(k) \subseteq (U - I))\}$$

$$IT = T - (LT \cup UT)$$

The set S_k includes the blocks required by a given transaction k; it is

$$S_k = \{L\} \text{ if } k \in LT$$

$$S_k = \{U\} \text{ if } k \in UT$$

$$S_k = \{L, U\} \text{ if } k \in IT$$

The counts CL and CU weigh all transactions that need to access either block L or block U; the count CI weighs transctions that need to access both blocks.

$$CL = \sum_{k \in LT} \text{acc}_k$$

$$CU = \sum_{k \in UT} \text{acc}_k$$

$$CI = \sum_{k \in IT} \text{acc}_k$$

In the multiple site problem, we use F to denote a fragment being allocated and alloc_F to denote the allocation sites for fragment F; alloc_F may contain either a single site in nonreplicated VPP or multiple sites in the replicated VPP.

3.2 The VPP for a Single Site with One Memory Level

The partitioning of an object on a single site with homogeneous devices requires the repeated use of the SPLIT algorithms, introduced in Sections 2.3, 2.4, and 2.6.

The specific weight factors for this problem, to be used in the objective function (3), are

$$w1 = \sum_{k} \text{acc}_k \sum_{F \in S_k} \sum_{(i \in F) \wedge (i \notin A(k))} l_i$$

$$w2 = CL + CU + 2 \times CI$$

$$w3 = \sum_{i \in I} l_i$$

The weight $w1$ sums the length of irrelevant attributes accessed by transactions; it evaluates the inefficiency of the transfer from mass memory to main memory, where only relevant attributes will be used.

The weight $w2$ counts the number of accesses made by transactions to all the partitions; it evaluates the number of input–output operations that are required, assuming that this number is proportional to the total number of the records accessed.

The weight $w3$ counts the amount of memory that is necessary to store the overlapping attributes, which are replicated on both partitions.

Figure 12 shows an application of the VPP for a single site, overlapping partitioning with one memory level which uses the goal function (3); the cost factors were set as follows: $C1 = 1$, $C2 = 1000$, $C3 = 100$. In the figure, the result of all possible combinations of use of the SHIFT and CLUSTER procedures is shown. Notice that the use of CLUSTER reduces the number of partitions in the result.

3.3 The VPP for a Single Site with Multiple Memory Levels

The partitioning of an object between a fast (and costly) and a slow (and cheap) memory device requires the repeated use of the PRI_SEC algorithm, described in Sections 2.5 and 2.6. The cost $C1$ is deemed irrelevant to this problem; the

Problem type: S-O--	Attribute	5	1	7	2	8	3	9	10	4	6	Iteration
	Part											level
	1	X	X	X								2
	2			X	X							2
	3			X	X	X						2
	4				X	X	X	X	X	X	X	2

Problem type: S-OS-	Attribute	5	1	7	2	8	3	9	10	4	6	Iteration
	Part											level
	1	X	X	X								2
	2			X	X							2
	3			X	X	X						2
	4				X	X	X	X	X	X	X	2

Problem type: S-O-C	Attribute	6	10	4	9	3	8	5	7	1	2	Iteration
	Part											level
	1	X	X	X			X		X	X	X	2
	2							X	X	X	X	2
	3				X	X	X	X	X	X	X	1

Problem type: S-OSC	Attribute	5	1	8	9	10	4	7	6	2	3	Iteration
	Part											level
	1	X	X					X				2
	2							X		X		2
	3			X	X	X	X	X	X	X	X	1

Fig. 12. The use of the objective function (3) in the SPLIT_OVERLAP problem, combining in all possible ways the use of SHIFT and CLUSTER procedures.

weight factors to be used in the objective function (3) are

$$w2 = CL$$

$$w3 = \sum_{i \in U} l_i$$

The weight $w2$ counts the number of accesses made by transactions to the secondary memory, while $w3$ counts the amount of data in the fast memory.

A generalization of the primary–secondary memory allocation problem, the *memory hierarchy allocation problem* (MHAP), is defined as follows: "Partition an object into m levels of memory such that data is allocated from level 1 to level m, 1 being the costliest and m being the cheapest memory, so as to minimize the total cost of storage and processing for that object."

The proposed solution method consists of invoking PRI_SEC $m - 1$ times; each time a pair of memories is considered, starting from the two levels m and $m - 1$. After having solved the first problem, the primary partition is temporarily assigned to memory $m - 1$ and the secondary partition is assigned to memory m. Then, memories $m - 2$ and $m - 1$ are considered, and the fragment assigned to memory $m - 1$ is evaluated for further partitioning. If at some iteration i no partition is assigned to the memory at level $i - 1$, the solution method proceeds considering the two noncontiguous levels of i and $i - 2$. The procedure terminates when level 1 is reached.

Problem type: H	Attribute	3	9	2	8	7	5	1	10	4	6	Iteration level
levels 1 and 2	PRI	X	X	X	X	X						1
	SEC						X	X	X	X	X	
levels 2 and 3	PRI	X	X	X	X							1
	SEC					X						

Fig. 13. The use of the PRI_SEC algorithm to solve the MHAP problem.

Clearly, each partitioning problem is characterized by a different value of the coefficients $C2$ and $C3$, which reflect the differences in price and performances of the considered pairs of memories. Figure 13 shows an example of the MHAP problem with three levels. The cost factors between level 1 and 2 were set to $C1 = 100$, $C2 = 1$; those between levels 2 and 3 were set to $C1 = 20$, $C2 = 3$. Note that at each iteration for the given two levels of memory it is important to consider the ratio of costs, and not the absolute cost.

3.4 The VPP on Multiple Sites with Nonreplicated Allocation

The VPP on multiple sites with nonreplicated allocation is solved by a repetitive use of the MULTI_ALLOCATE_N algorithm described in Figure 14a. The algorithm applies to an object O and produces a nonoverlapping binary split into two blocks L and U; the splitting process is the same as in the SPLIT_NONOVERLAP algorithm, and can take advantage of the use of SHIFT or CLUSTER algorithms as described before.

For each possible partition L and U, the algorithm attempts all possible fragment allocations (i.e., m^2 cases for m sites) and selects the pair with the least cost. The weight factors $w1$, $w2$, and $w3$ are computed in the same way as for the single-site, homogeneous devices VPP (see Section 3.2). However, in a multiple-site problem the transmission cost $C4$ is predominant; the weight factor $w4$ is given by

$$w4 = \sum_k \sum_{F \in S_k} \sum_{j \mid \text{alloc}_F \neq j} n_{kj} \text{freq}_{kj} \sum_{i \in (F \cap A(k))} l_i.$$

The weight $w4$ counts the number of bytes selected from records on remote sites that need to be transmitted to the transaction site.[3]

A table of partitions versus allocations is maintained, which stores the allocation sites for each fragment produced by the algorithm. If an object O is split into two objects L and U by the algorithm, then the entry for O is deleted, and two entries for L and U are inserted. The process of splitting is repeated until no partition may be split with any further gain.

Figure 15a shows the solution of the nonreplicated, multiple-site VPP problem, whose input parameters are specified in Tables I and II. Cost factors were set as follows: $C_1 = 1$, $C_2 = 100$, $C_3 = 10$, and $C_4 = 1000$.

[3] This simple transaction model applies to typical applications using distributed file systems or unsophisticated distributed databases, where the system collects the information at the site of the transaction and then executes the transaction there. A more complicated transaction model, which is more realistic for advanced systems, is in [4].

```
procedure MULTI_ALLOCATE_N (0)
  begin
  for all points x on the diagonal of clustered matrix AA (determining L and U) do
    for all possible allocations j1 of L do
      for all possible allocations j2 of U do
        begin
        evaluate z(L, U, j1, j2)
        if there is improvement then remember L, U, j1, j2
        end
  if there was improvement then
    begin
    delete ⟨0, j0⟩
    store ⟨L, j1⟩
    store ⟨U, j2⟩
    MULTI_ALLOCATE_N (L)
    MULTI_ALLOCATE_N (U)
    end
  end
```

Table of partitions and allocations

Partition	Allocation
0	$j0$

Fig. 14(a). Algorithm MULTI_ALLOCATE_N.

```
procedure MULTI_ALLOCATE_R(F)
  begin
  start from alloc_F = nonreplicated solution of MULTI_ALLOCATE_N
  repeat
    try all sites not in alloc_F
    let j be the most beneficial
    alloc_F = alloc_F ∪ j
  until
    no further benefit
  end
```

Fig. 14(b). Algorithm MULTI_ALLOCATE_R.

3.5 The VPP on Multiple Sites with Replicated Allocation

To consider the replicated allocation in the multiple-site case, we first solve the problem of nonreplicated partitioning using the algorithm MULTI_ALLOCATE_N; then we apply the algorithm MULTI_ALLOCATE_R to each fragment in the solution of that problem independently. The algorithm MULTI_ALLOCATE_R is shown in Figure 14b.

The MULTI_ALLOCATE_R algorithm applies to each normalized fragment F individually, and proceeds by allocating it to the sites that are not in alloc_F, until no further benefit is obtained. The algorithm can be classified as a "greedy" heuristic, as at each iteration the most convenient site is selected for replication.

The objective function considers only those costs that depend on the degree of replication of a single fragment F; cost $C1$ is irrelevant, and the following weight

Problem type: MNN—3	Attribute	5	1	7	2	8	3	9	10	4	6	Allocation 1	2	3	4	Depth level
	Part															
	1	X	X	X	X	X	X	X					X			1
	2								X	X	X				X	1

Fig. 15(a). Results of the application of the algorithm MUTLI_ALLOCATE_*N*.

Problem type: MRN—3	Attribute	5	1	7	2	8	3	9	10	4	6	Starting allocation 1	2	3	4	Final allocation 1	2	3	4
	Part																		
	1	X	X	X	X	X	X	X					X				X		
	2								X	X	X				X			X	X

Fig. 15(b). Results of the application of the algorithm MULTI_ALLOCATE_*R*.

factors are computed:

$$w2 = \sum_{k \mid (\text{type}_k = \text{'u'}) \wedge (A(k) \cap F \neq \Phi)} \sum_{j} n_{kj} \text{freq}_{kj}$$

$$w3 = \sum_{i \in P} |\text{alloc}_F| \sum_{i \in P} l_i$$

$$w4 = \sum_{k \mid A(k) \cap F \neq \Phi} \sum_{j \mid j \notin \text{alloc}_F} n_{kj} \text{freq}_{kj} \sum_{i \in (F \cap A(k))} l_i.$$

The weight $w2$ counts the number of update operations that have to be applied to each copy of the replicated fragment in order to preserve their mutual consistency.

The weight $w3$ counts the amount of memory that is required for storing the replicated copies. The weight $w4$ counts the number of bytes that need to be transmitted to the site of the transaction, as in Section 3.4.

Notice that retrieval transactions are now likely to find a local copy of the required information, thus $w4$ decreases with the increase of replication; however, update and storage costs, represented by $w2$ and $w3$, grow with the number of copies, thus establishing a trade-off.

Figure 15b shows the results produced by applying the MULTI_ALLOCATE_R algorithm to the solution of the nonredundant allocation problem shown in Figure 15a. It shows that the algorithm finds it profitable to allocate an additional copy of fragment 2 at site 3.

4. CONCLUSIONS

In this paper we have presented several algorithms for determining vertical fragments and discussed their application to the design of databases. All the algorithms described in the paper have been implemented in PASCAL; the run for the final example shown in Figure 16, with 20 attributes and 15 variables, used only 1 second of CPU time on a DEC-2060 system.

Our approach is similar to that of Hoffer and Severance [13], in that we use the notion of affinity of attributes to permute them and produce an initial clustering. However, we have designed algorithms for an automatic selection of vertical fragments, which substitutes for the designer's subjective judgment, as proposed in [13].

	Attribute usage matrix																				Type	Single site frequency
Attribute	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20		
T 1	1	0	0	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1 = R	1 = 50
T 2	0	1	0	0	0	0	0	0	1	0	0	1	1	1	0	0	0	0	0	0	2 = R	2 = 50
T 3	0	0	1	0	0	0	1	0	0	1	1	0	0	0	0	0	1	1	0	0	3 = R	3 = 50
T 4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1	1	4 = R	4 = 50
T 5	1	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	5 = R	5 = 15
T 6	1	0	0	0	0	0	0	1	0	1	1	0	0	0	0	0	0	0	0	0	6 = R	6 = 15
T 7	0	0	1	0	0	0	1	0	0	1	1	0	0	0	0	0	1	0	0	0	7 = R	7 = 15
T 8	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	1	0	1	8 = R	8 = 15
T 9	0	1	0	0	1	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	9 = R	9 = 10
T10	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	1	0	0	10 = R	10 = 10
T11	1	1	1	0	1	1	0	0	1	0	0	1	1	0	0	0	0	0	0	0	11 = U	11 = 10
T12	0	0	0	1	0	0	1	0	0	1	0	0	0	1	0	0	0	0	1	1	12 = U	12 = 10
T13	0	0	0	0	0	0	0	1	0	0	1	0	0	0	1	1	1	1	0	0	13 = U	13 = 10
T14	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	14 = U	14 = 5
T15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	15 = U	15 = 5

Fig. 16(a). Parameters for the final example, with 20 attributes and 15 transactions.

Clustering of attributes on affinity matrix

Attribute	4	6	8	1	5	9	13	12	2	14	18	17	11	10	3	7	19	20	15	16
4	65	55	55	55	55	5	0	0	5	10	0	0	0	10	5	15	10	10	0	0
6	55	65	55	85	65	15	10	10	15	0	0	0	0	0	15	5	0	0	0	0
8	55	55	95	85	70	5	0	0	5	0	10	10	25	15	5	5	0	0	10	10
1	55	65	85	105	80	25	10	10	15	0	10	0	15	15	15	5	0	0	0	10
5	55	65	70	80	90	15	10	10	25	10	0	0	10	0	15	5	10	0	0	0
9	5	15	5	25	15	75	60	80	65	50	10	0	0	0	15	5	0	0	0	10
13	0	10	0	10	10	60	75	75	75	65	15	0	0	0	10	0	0	15	15	15
12	0	10	0	10	10	60	75	75	75	65	15	0	0	0	10	0	0	15	15	15
2	5	15	5	15	25	65	75	75	90	75	15	0	10	0	15	5	10	15	15	15
14	10	0	0	0	10	50	65	65	75	85	15	0	10	10	0	10	20	25	15	15
18	0	0	10	10	0	10	15	15	15	15	90	65	60	50	50	50	5	20	30	40
17	0	0	10	0	0	0	0	0	0	0	65	60	75	65	65	65	5	5	15	15
11	0	0	25	15	10	0	0	0	10	10	80	75	100	80	85	85	10	0	10	10
10	10	0	15	15	0	0	0	0	0	10	50	85	80	90	65	75	10	10	0	0
3	5	15	5	15	15	15	10	10	15	0	50	65	65	65	80	70	0	0	0	0
7	15	5	5	5	5	5	0	0	5	10	50	65	65	75	70	80	10	10	0	0
19	10	0	0	0	10	0	0	0	10	20	5	5	10	10	0	10	75	65	55	55
20	10	0	0	0	0	0	15	15	15	25	20	5	0	10	0	10	85	80	70	70
15	0	0	10	0	0	0	15	15	15	15	30	15	10	0	0	0	55	70	80	80
16	0	0	10	10	0	10	15	15	15	15	40	15	10	0	0	0	55	70	80	90

Problem type: S-N

Attribute Part	4	6	8	1	5	9	13	12	2	14	18	17	11	10	3	7	19	20	15	16	Iteration level
1	X	X	X	X	X																1
2						X	X	X	X	X											2
3											X	X	X	X	X	X					3
4																	X	X	X	X	3

Fig. 16(b). Solution of the final example (single site, nonoverlapping VPP).

The algorithms do not require very sophisticated input data. In fact, according to the well-known 20–80 rule, a limited number of important transactions (20 percent) typically account for most of the use of the database (80 percent), and therefore it is sufficient to collect input data for them. Moreover, we have kept the required information about the database usage to a minimum; we require only the specification as to whether an attribute is used by a transaction and the number of logical accesses to the object being considered.

We postulate the integration of these algorithms into an interactive design tool. With such a tool, the designer will typically be able to select a proper mix of algorithms and to use partial results to steer the design appropriately. This tool is applied for the initial design of the database, or for the reconfiguration of the database at an intermediate stage of its life cycle.

Further extensions of this work will be in the direction of developing site-specific affinity measures for distributed databases, of incorporating different transaction processing strategies, and of considering new applications of vertical partitioning, such as parallel processing of transactions.

ACKNOWLEDGMENTS

Many colleagues have participated in discussions on vertical partitioning, and have inspired some of the concepts used here. Particular thanks are due to Sheldon Finkelstein, Arthur Keller, Don Batory, and to the anonymous referees.

REFERENCES

1. BABAD, M.J. A record and file partitioning model. *Commun. ACM 20*, 1 (Jan. 1977).
2. BERNSTEIN, P.A., GOODMAN, N., WONG, E., REEVE, C.L., AND ROTHNIE, J.B., JR. Query processing in a system for distributed databases (SDD-1). *ACM Trans. Database Syst. 6*, 4 (Dec. 1981), 602–625.
3. CERI, S., NEGRI, M., AND PELAGATTI, G. Horizontal data partitioning in database design. In *Proceedings ACM SIGMOD International Conference on Management of Data* (Orlando, Fla., 1982), ACM, New York.
4. CERI, S., NAVATHE, S.B., AND WIEDERHOLD, G. Distribution design of logical database schemas. *IEEE Trans. Softw. Eng. SE-9*, 3 (July 1983).
5. CERI, S., AND PELAGATTI, G. The allocation of operations in distributed database access. *IEEE Trans. Comput. TC-32*, 2 (1982).
6. CHANG, S.K., AND CHENG, W.H. A methodology for structured database decomposition. *IEEE Trans. Softw. Eng. SE-6*, 2 (Mar. 1980).
7. CHEN, P.P.S., AND AKOKA, J. Optimal design of distributed information systems. *IEEE Trans. Comput. C-29*, 12 (Dec. 1980).
8. DAYAL, U., AND BERNSTEIN, P.A. The fragmentation problem: Lossless decomposition of relations into files. Tech. Rep. CCA-78-13, Computer Corporation of America, Cambridge, Mass.
9. EISNER, M.J., AND SEVERANCE, D.G. Mathematical techniques for efficient record segmentation in large shared databases. *J. ACM 23*, 4 (Oct. 1976).
10. FISHER, M.L., AND HOCHBAUM, D.S. Database location in computer networks. *J. ACM 27*, 4 (Oct. 1980), 718–735.
11. HAMMER, M., AND NIAMIR, B. A heuristic approach to attribute partitioning. In *Proceedings ACM SIGMOD International Conference on Management of Data* (Boston, Mass., 1979), ACM, New York.
12. HOFFER, J.A. An integer programming formulation of computer database design problems. *Inf. Sci. 11* (July 1976), 29–48.
13. HOFFER, J.A., AND SEVERANCE, D.G. The use of cluster analysis in physical database design. In *Proceedings 1st International Conference on Very Large Databases* (Framingham, Mass., 1975).
14. LUM, V.Y. The 1978 New Orleans Database Design Workshop Report. In *Proceedings 5th International Conference on Very Large Databases* (Rio de Janeiro, Oct. 1979).
15. MAHMOUD, S., AND RIORDON, J.S. Optimal allocation of resources in distributed information networks. *ACM Trans. Database Syst. 1*, 1 (1976).
16. MARCH, S.T., AND SEVERANCE, D.G. The determination of efficient record segmentations and blocking factors for shared data files. *ACM Trans. Database Syst. 2*, 3 (Sept. 1977).
17. MCCORMICK, W.T., SCHWEITZER, P.J., AND WHITE, T.W. Problem decomposition and data reorganization by a clustering technique. *Oper. Res. 20*, 5 (Sept. 1972), 993–1009.
18. MORGAN, H.L., AND LEVIN, J.D. Optimal program and data location in computer networks. *Commun. ACM 20*, 1 (Jan. 1977).
19. ROTHNIE, J.B., JR., ET AL. Introduction to a system for distributed databases (SDD-1). *ACM Trans. Database Syst. 6*, 4 (Dec. 1980), 431–466.
20. SCHKOLNICK, M. A clustering algorithm for hierarchical structure. *ACM Trans. Database Syst. 2*, 1 (Mar. 1977).
21. SELINGER, P.G., AND ADIBA, M.E. Access path selection in distributed database management systems. In *Proceedings of the International Conference on Databases* (Aberdeen, Scotland, July 1980), British Computer Society.
22. YAO, S.B., NAVATHE, S.B., AND WELDON, J.L. An integrated approach to logical database design. In *Proceedings NYU Symposium on Database Design, 1978.* Lecture Series in Computer Science, No. 132, Springer-Verlag, New York, 1982.
23. WIEDERHOLD, G. *Database Design.* 2nd ed., McGraw-Hill, New York, 1983.
24. WILLIAMS, R., ET AL. R*, an overview of the architecture. In *Proceedings 2nd International Conference on Databases: Improving Usability and Responsiveness.* P. Scheuermann, Ed., Academic Press, New York, 1981.

Received January 1983; revised August 1983; accepted February 1984