Vertical Partitioning for Database Design: A Graphical Algorithm

Shamkant B. Navathe and Minyoung Ra

Database Systems Research and Development Center
Computer and Information Science Department
University of Florida
Gainesville, Florida 32611

ABSTRACT

Vertical partitioning is the process of subdividing the attributes of a relation or a record type, creating fragments. Previous approaches have used an iterative binary partitioning method which is based on clustering algorithms and mathematical cost functions. In this paper, however, we propose a new vertical partitioning algorithm using a graphical technique. This algorithm starts from the attribute affinity matrix by considering it as a complete graph. Then, forming a linearly connected spanning tree, it generates all meaningful fragments simultaneously by considering a cycle as a fragment. We show its computational superiority. It provides a cleaner alternative without arbitrary objective functions and provides an improvement over our previous work on vertical partitioning.

1. Introduction

The partitioning of a global schema into fragments can be performed in two different ways: *vertical partitioning* and *horizontal partitioning* [Ceri 84]. This paper is concerned with vertical partitioning.

Vertical partitioning is the process that divides a *global object* which may be a single relation or more like a universal relation into groups of their attributes, called vertical fragments [Nava 84, Nava 85, Corn 87]. It is used during the design of a database to enhance the performance of transactions [Nava 84]. In order to obtain improved performance, fragments must closely match the requirements of the transactions. Vertical partitioning has a variety of applications wherever the *match* between data and transactions can affect performance. That includes partitioning of individual files in centralized environments, data distribution in distributed databases, dividing data among different levels of memory hierarchies, and so on.

Hoffer and Severance [Hoff 75] measure the affinity between pairs of attributes and try to cluster attributes according to their pairwise affinity by using the bond energy algorithm (BEA) developed in [McCo 72].

Navathe, et. al. [Nava 84] extend the results of Hoffer and Severance and propose a two phase approach for vertical partitioning. During the first phase, they use the given input parameters in the form of an attribute usage matrix and transactions, to construct the attribute affinity matrix on which clustering is performed. After clustering, iterative binary partitioning is attempted, first with an empirical objective function. The process is continued until no further partitioning results. During the second phase, the fragments can be further refined by incorporating estimated cost factors weighted on the basis of the type of problem being solved.

Cornell and Yu [Corn 87] apply the work of [Nava 84] to relational databases. They propose an algorithm which decreases the number of disk accesses to obtain an optimal binary partitioning. They show how knowledge of specific physical factors may be incorporated into the fragmentation methodology to yield better overall performance.

Ceri, Pernici and Wiederhold [Ceri 88] extend Navathe, et. al.'s work by considering it as a DIVIDE tool and by adding a CONQUER tool. Their CONQUER tool again extends the same basic approach in the direction of adding details about operations and physical accesses similar to [Corn 87]. This approach is focussed on the decomposition of the design process into several design subproblems and provides no algorithmic improvement in the process of vertical partitioning itself.

In all algorithms that we have surveyed, the binary partitioning technique has been used for partitioning *after clustering* attributes. Thus binary partitioning is required to be repeated until all meaningful fragments are determined. It is also necessary that clustering be repeated at each iteration after clustering two new affinity matrices corresponding to the newly generated fragments.

In this paper we propose a new vertical partitioning algorithm which has less computational complexity and generates all meaningful fragments simultaneously by using a graphical method. This approach is based on the fact that all pairs of attributes in a fragment have high "*within fragment affinity*" but low "*between fragment affinity*". Section 2 deals with preliminaries. In Section 3, the algorithm and its analysis are presented. Section 4 describes the application of the proposed approach, and Section 5 gives the conclusion.

2. Preliminaries

2.1 Overview

The algorithm that we propose starts from the *attribute affinity (AA) matrix*, which is generated from the attribute usage matrix using the same method as that of our previous approach [Nava 84]. The *attribute usage matrix* represents the use of attributes in *important* transactions. Each row refers to one transaction; the "1" entry in a column indicates that the transaction "uses" the corresponding attributes. Whether the transaction retrieves or updates the relation can also be captured by another column vector with R and U entries for retrieval and update. That information may be used by an empirical objective function as in [Nava 84]. The attribute usage matrix for 10 attributes and 8 transactions is shown in Figure 1. Attribute affinity is defined as

Attribute usage matrix											Type	Number of accesses per time period
Attributes / Transactions	1	2	3	4	5	6	7	8	9	10		
T1	1	0	0	0	1	0	1	0	0	0	R	Acc 1 = 25
T2	0	1	1	0	0	0	0	1	1	0	R	Acc 2 = 50
T3	0	0	0	1	0	1	0	0	0	1	R	Acc 3 = 25
T4	0	1	0	0	0	0	1	1	0	0	R	Acc 4 = 35
T5	1	1	1	0	1	0	1	1	1	0	U	Acc 5 = 25
T6	1	0	0	0	1	0	0	0	0	0	U	Acc 6 = 25
T7	0	0	1	0	0	0	0	0	1	0	U	Acc 7 = 25
T8	0	0	1	1	0	1	0	0	1	1	U	Acc 8 = 15

Fig.1 Attribute usage matrix

Attributes	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

Fig.2 Attribute affinity (AA) matrix

$$aff_{ij} = \sum_{k \in \tau} acc_{kij}$$

where acc_{kij} is the number of accesses of transaction k referencing both attributes i and j. The summation occurs over all transactions that belong to the set of important transactions τ. This definition of attribute affinity measures the strength of an imaginary bond between the two attributes, predicated on the fact that attributes are used together by transactions. Based on this definition of attribute affinity, the attribute affinity matrix is defined as follows: It is an n x n matrix for the n-attribute problem whose (i,j) element equals aff_{ij}. Figure 2 shows the attribute affinity matrix which was formed from the Figure 1. A diagonal element AA(i,i) equals the sum of the elements in the attribute usage matrix for the column which represents a_i. This is reasonable since it shows the "strength" of that attribute in terms of its use by all transactions.

A note about the attributes: in this proposed technique as well as in the previous techniques, the set of attributes considered may be

(a) the universal set of attributes in the whole database.

(b) the set of attributes in a single relation (or record type).

By using (a), the fragments generated may be interpreted as relations or record types. By using (b), fragments of a single relation are generated.

In previous approaches, they apply a clustering algorithm to the AA matrix. In our present approach, however, we consider the AA matrix as a complete graph called the *affinity graph* in which an edge value represents the affinity between the two

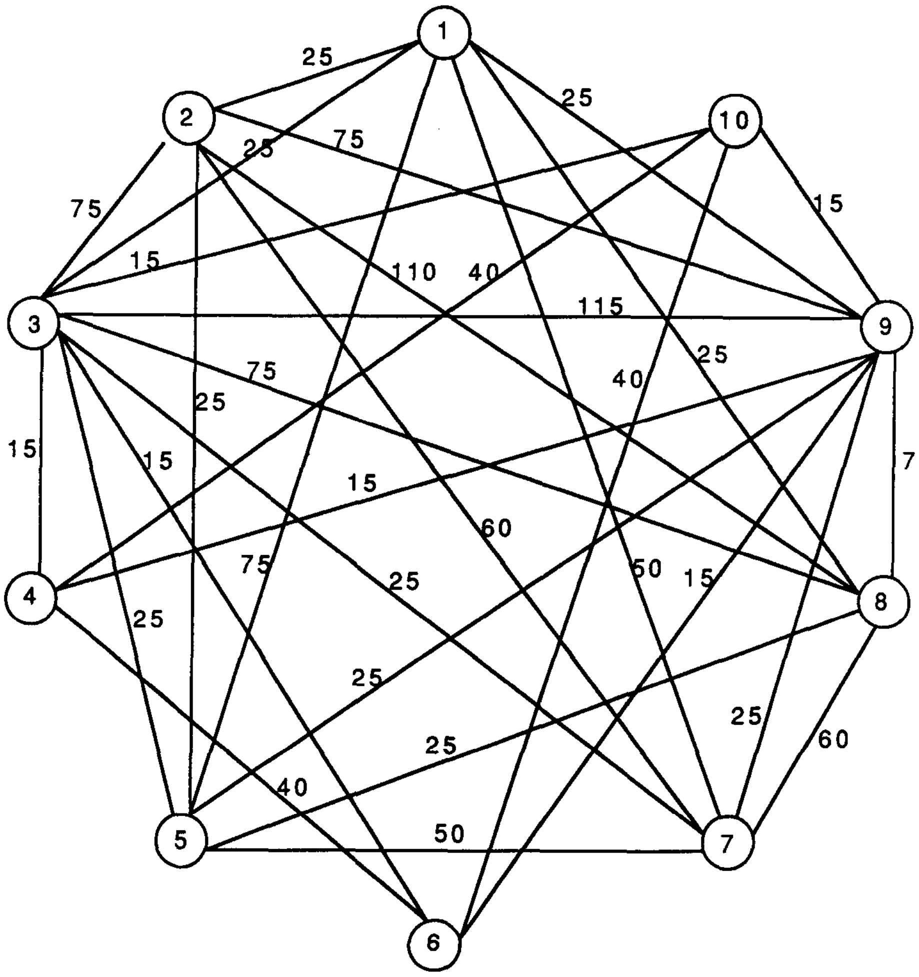

Fig. 3 Affinity graph after excluding zero-valued edges

attributes. Then, forming a linearly connected spanning tree, the algorithm generates all meaningful fragments in one iteration by considering a cycle as a fragment. A "*linearly connected*" tree has only two ends. Figure 3 shows the affinity graph corresponding to the AA matrix of Figure 2 after excluding zero-valued edges. Note that the AA matrix serves as a data structure for the affinity graph.

The major advantages of the proposed method over that in [Nava 84] are that:

(a) There is no need for iterative binary partitioning. The major weakness of iterative binary partitioning is that at each step two new problems are generated increasing the complexity; furthermore, termination of the algorithm is dependent on the discriminating power of the objective function.

(b) The method requires no objective function. The empirical objective functions in [Nava 84] were selected after some trial and error experimentation to see that they possess a good discriminating power. Although reasonable, they constitute an arbitrary choice. This arbitrariness has been eliminated in the proposed methodology.

2.2 Definitions and notations

We shall use the following notation and terminology in the description of our algorithm.

- A,B,C,... denotes nodes.
- a,b,c,... denotes edges.
- p(e) denotes the affinity value of an edge e.
- *primitive cycle* denotes any cycle in the affinity graph.
- *affinity cycle* denotes a primitive cycle that contains a cycle node. In this paper we assume that a cycle means an affinity cycle, unless otherwise stated.
- *cycle completing edge* denotes a "to be selected" edge that would complete a cycle.
- *cycle node* is that node of the cycle completing edge, which was selected earlier.
- *former edge* denotes an edge that was selected between the last cut and the cycle node.
- *cycle edge* is any of the edges forming a cycle.
- *extension of a cycle* refers to a cycle being extended by pivoting at the cycle node.

The above definitions are used in the proposed algorithm to process the affinity graph and to generate possible cycles from the graph. They will become clearer when we explain them in Section 2.3. Each cycle gives rise to a vertical fragment. The intuitive explanation of why such a procedure yields meaningful fragments can be given only after we fully describe the algorithm in Section 3 of the paper.

2.3 Fundamental concepts

Based on the above definitions we would like to explain the mechanism of forming cycles. For example, in Figure 4, suppose edges a and b were selected already and c was selected next. At this time, since c forms a primitive cycle, we have to check if it is an affinity cycle. This can be done by checking the possibility of a cycle. *Possibility of a cycle* results from the condition that *no former edge exists, or p(former edge) <= p(all the cycle edges)*. The primitive cycle a,b,c is an affinity cycle because it has no former edge and satisfies the possibility of a cycle. Therefore the primitive cycle a,b,c is marked as a candidate partition and node A becomes a cycle node.

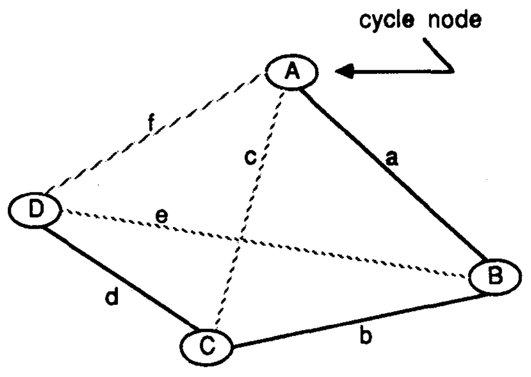

Fig. 4 Cycle and extension

Now let us explain how the extension of a cycle is performed. In Figure 4, after the cycle node is determined, suppose edge d was selected. At this time, d is checked as a potential edge for extension. It can be done by checking the possibility of extension of the cycle by d. *Possibility of extension* results from the condition of *p(edge being considered or cycle completing edge) >= p(any one of the cycle edges)*. Thus the old cycle a,b,c is extended to the new cycle a,b,d,f if the edge d under consideration, or the cycle completing edge f, satisfies the possibility of extension which is: p(d) or p(f) >= minimum of (p(a),p(b),p(c)). Now the process is continued: suppose e was selected as the next edge. But we know from the definition of the *extension of a cycle* that e cannot be considered as a potential extension because the primitive cycle d,b,e does not include the cycle node A. Hence it is discarded and the process is continued.

The next concept that we wish to explain corresponds to the relationship between a cycle and a partition. There are two cases in partitioning.

(1) Creating a partition with a new edge.

In the event that the edge selected next for inclusion (e.g. d in Figure 4) was not considered before, we call it a *new edge*. If a new edge by itself does not satisfy the possibility of extension, then we continue to check an additional new edge called cycle completing edge (e.g. f in Figure 4) for the possibility of extension. In Figure 4, new edges d and f would potentially provide such a possibility of extension of the earlier cycle formed by edges a,b,c.

If d,f meet the condition for possibility of extension stated above (namely p(d) or p(f) >= minimum of (p(a),p(b),p(c))), then the extended new cycle would contain edges a,b,d,f. If the condition were not met, we produce a cut on edge d (called the *cut edge*) isolating the cycle a,b,c. This cycle can now be considered a *partition*.

(2) Creating a partition with a former edge.

After cutting in (1), if there is a former edge, then change the previous cycle node to that node where the cut edge was incident, and check for the possibility of extension of the cycle by the former edge. For example, in Figure 5, suppose that a,b, and c form a cycle with A as the cycle node, and that there is a cut on d, and that the former edge w exists. Then the cycle node A is changed to C because the cut edge d originates in C. We are now evaluating the possibility of extending the cycle a,b,c into one that would contain the former edge w. Hence we consider the possibility of the cycle a,b,e,w. Assume that w or e does not satisfy the possibility of extension, i.e., if "p(w) or p(e) >= minimum of (p(a),p(b),p(c))" is not true. Then the result is the following: i) w will be declared as a cut edge, ii) C remains as the cycle node, and iii) a,b,c becomes a partition. Alternately, if the possibility of extension is satisfied, the result is: i) cycle a,b,c is extended to cycle w,a,b,e, ii) C remains as the cycle node, and iii) *no* partition can yet be formed.

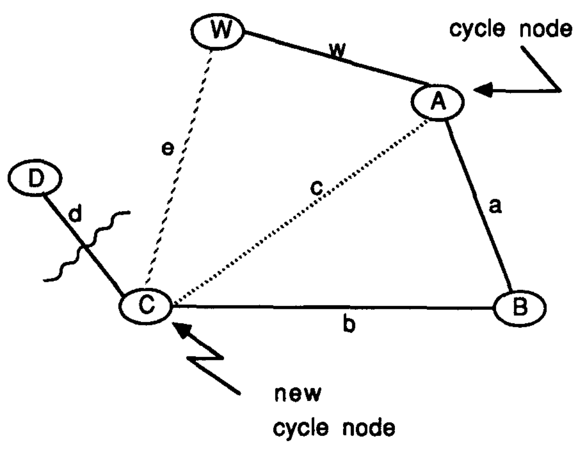

Fig. 5 Partition

Intuitively, the algorithm presented below achieves the decision of partitioning in the following manner. Keeping the pivot on a present cycle node, extension of the cycle is attempted by considering either new edges or former edges which would expand the *area under the cycle*. For example, in Figure 4 we attempted to "grow the cycle" from area ABC to area ABCD by considering new edges d and f. In Figure 5, we shifted the pivot from node A to C and then attempted to grow from area ABC to ABCW with a former edge w. The next growth of ABCW would be attempted counterclockwise with respect to the pivoting cycle node C by considering edges "*former*" to w incident on node W.

3. The algorithm

An algorithm for generating the vertical fragments by the affinity graph is described below. Each partition of the graph generates a vertical fragment.

3.1 Description of the algorithm

First we briefly describe the algorithm in 5 steps.

Step 1. Construct the affinity graph of the attributes of the object being considered. Note that the AA matrix is itself an adequate data structure to represent this graph. No additional physical storage of data would be necessary.

Step 2. Start from *any node.*

Step 3. Select an edge which satisfies the following conditions:
- It should be linearly connected to the tree already constructed.
- It should have the largest value among the possible choices of edges at each end of the tree.

This iteration will end when all nodes are used for tree construction.

Step 4. When the next selected edge forms a primitive cycle:
(1) If a cycle node does not exist, check for the "possibility of a cycle" and if the possibility exists, mark the cycle as an affinity cycle. Consider this cycle as a candidate partition. Go to step 3.
(2) If a cycle node exists already, discard this edge and go to step 3.

Step 5. When the next selected edge does not form a cycle and a candidate partition exists:
(1) If no former edge exists, check for the possibility of extension of the cycle by this new edge. If there is no possibility, cut this edge and consider the cycle as a partition. Go to step 3.
(2) If a former edge exists, change the cycle node and check for the possibility of extension of the cycle by the former edge. If there is no possibility, cut the former edge and consider the cycle as a partition. Go to step 3.

To obtain a more detailed algorithm, suppose that the following data structures are used during implementation [Bras 88]: The nodes of the affinity graph are numbered from 1 to n, N = {1,2, ..., n}, and a symmetric matrix L gives the weight of each edge. Three vectors are used: B, strongest, and maxwt. B gives the sequence of scanned nodes. For each node $i \in N \setminus B$, strongest[i] gives the node in B that is strongest with respect to i, and maxwt[i] gives the weight from i to strongest[i]; strongest[1] and maxwt[1] are not used. Without loss of generality we can assume that the algorithm starts from node 1. The detailed description of the algorithm now follows. The algorithm uses variables with the following meaning:

- *p_cycle*: is a binary variable which denotes whether a primitive cycle exists.
- *c_node*: is a binary variable which denotes whether a cycle node exists.
- *f_edge*: is a binary variable which denotes whether a former edge exists.
- *candidate_p*: is a binary variable which denotes whether an affinity cycle exists which can potentially generate a fragment.
- *cycle_c_edge_wt*: is an integer variable for the weight of the cycle completing edge.
- *former_edge_wt*: is an integer variable for the weight of the former edge.

```
Procedure  Make_partition(L[1..n,1..n]): set of edges
  {initialize flags and variables}
  B[1] <- 1
  f1 <- 1, f2 <- 0    { f1 & f2 each refer to an end
                        of the spanning tree}
  p_cycle, c_node, f_edge, candidate_p <- false
  pmin <- ∞           {minimum edge of a cycle}
  {initialize vectors}
  for i <- 2 to n do
    strongest[i] <- 1
    maxwt[i] <- L[i,1]
    B[i] <- 0
  end_for
  repeat n-1 times
    max <- -1
    {select the next node}
    for j <- 2 to n do
      if    maxwt[j] > max and
              (strongest[j] = f1 or strongest[j] = f2)
      then  max <- maxwt[j]
              k <- j
    end_for
    {adjust the pointers for checking a primitive cycle}
    if strongest[k] = f1 then if f2 = 0 then f2 <- k
                                        else swap f1,f2
                                             f2 <- k
                         else f2 <- k
    {check if there is a primitive cycle}
    for j <- 2 to n do
      if    B[j] = k and c_node = false
      then p_cycle <- true
             if B[j-1] > 0 then f_edge <- true
    end_for
    if    p_cycle = true      {primitive cycle exists}
    then              {check if it is an affinity cycle}
        if    pmin >= former_edge_wt or f_edge = false
        then candidate_p <- true
              c_node <- true
        else  f_edge <- false
    else
        insert k into B, maxwt[k] <- - ∞
        if     candidate_p = true
        then if     maxwt[k] < pmin or
                    cycle_c_edge_wt < pmin
```

```
            then      {partition exists}
                      reinitialize variables
                      if     f_edge = false
                      then  save this partition
                      else  change the cycle node
                            if     former_edge_wt < pmin
                            then  save this partition
                            else  extend the cycle
            else  extend the cycle
     {pmin contains the minimum edge of a cycle}
     if pmin > maxwt[k] then pmin <- maxwt[k]
     {rearrange vectors for next selection}
     for j <- 2 to n do
        if L[k,j] > maxwt[j] then maxwt[j] <- L[k,j]
                                   strongest[j] <- k
     end_for
   end_repeat
```

3.2 Why does the above algorithm produce reasonable partitioning ? (an intuitive explanation)

Now we will give the proof of the correctness of the algorithm. The idea of the proof consists in showing that an affinity cycle is distinguished from other cycles in terms of the values of affinities. In other words, it means that all edges in a cycle should have *similar* affinities in contrast to other cycles.

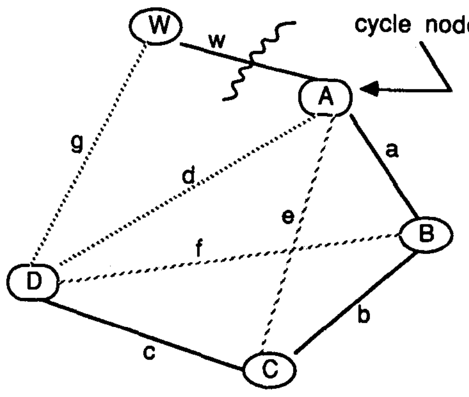

Fig.6 Proof of reasonable partitioning

In Figure 6, suppose that a,b,c and d initially form a cycle and that there is a cut on w. Then by the definition of the possibility of a cycle,

$p(a), p(b), p(c), p(d) >= p(w)$.

Now, consider the subcycle a,b and e. Since the cycle node A is included in this cycle,

$p(e) >= p(w)$.

Likewise,

$p(f) >= p(w)$.

Then, because the *next edge* which was selected first at node W was w, it implies that w was the largest

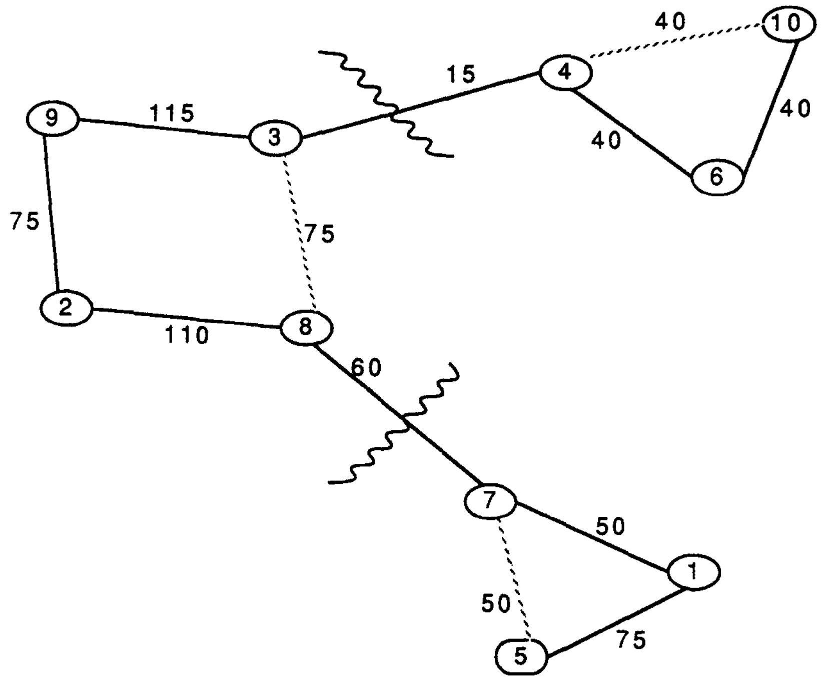

Fig. 7(a) Result of the first example: start at node 9 edge at W. Hence

p(w) >= p(g).

Thus

p(a),p(b),p(c),p(d),p(e),p(f) >= p(w) >= p(g).

At this time, if p(w) had been equal to any edge in the cycle, then the cycle would have been extended. Thus we can conclude that

p(a),p(b),p(c),p(d),p(e),p(f) > p(w).

This means that all edges in the cycle a,b,c,d have relatively similar affinities and are distinguished from other edges (namely, w and g) which are not in the cycle.

3.3 Examples

We will use the same example problems from [Nava 84] to illustrate how this algorithm works and to compare partitioning decisions. Since our algorithm uses the same attribute affinity matrix, we assume that it has already been completed from the original transaction matrix and the computation of affinities. For ease of understanding, we will refer back to the steps of the algorithm from Section 3.1.

The attribute affinity matrix of the first example is shown in Figure 2 and its affinity graph after excluding the zero-valued edges appears in Figure 3. Suppose we start at node 9 (step 2), then, by the algorithm, edges 9-3, 9-2, and 2-8 are selected in order (step 3). At this time, edge 8-9 cannot form a cycle because it does not satisfy the possibility of a cycle (step 3). Thus edge 8-3 is selected as the next edge and it forms a candidate partition (step 4). Note that node 3 becomes a cycle node (step 4). Then the process is continued and edge 8-7 is selected (step 3). Since there is a candidate partition, the possibility of extension is checked (step 5.1). Thus the cycle 9,3,2,8 considered as a partition because edge 8-7 (edge being considered) and 3-7 (cycle completing edge) are both less than any of the cycle edges (step 5.1). The relevant part of the graph is shown again in Figure 7(a). As shown in Figure 7(a), our algorithm generates three affinity cycles separated by edges 3-4 and 7-8. They generate three fragments: (1,5,7), (2,3,8,9), (4,6,10). From that Figure, we know that the result of our algorithm is the same as that of [Nava 84].

To show that this algorithm *does not depend upon the starting node*, let us start at node 1. By the algorithm, the first affinity cycle is not formed until edges 1-5, 5-7, 7-8, 8-2, 2-9, and 9-3 are selected. The first cycle 8,2,9,3 is identified as a candidate partition

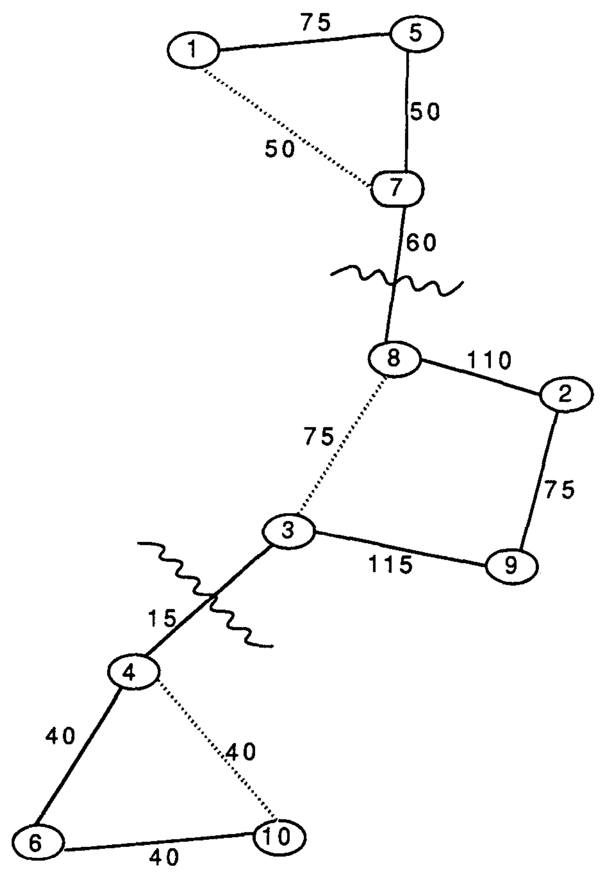

Fig. 7(b) Result of the first example: start at node 1

and node 8 becomes a cycle node. Then a cut occurs on edge 3-4 because neither edge 3-4 nor edge 4-8 satisfies the possibility of extension of the cycle (step 5.1). At this time, since there is a former edge, we have to change the cycle node to node 3 and check for the possibility of extension of the cycle by the former edge 7-8 (step 5.2). Thus another cut occurs on edge 7-8 because edge 7-8 and 7-3 are both less than any of the cycle edges. Figure 7(b) shows this result. Thus we can conclude that the resulting fragments are always the same irrespective of the node from which you start.

The second example we will use is a global relation with 20 attributes and 15 transactions. The result of [Nava 84] partitions this relation into four fragments in three iterations: (1,4,5,6,8), (2,9,12,13,14), (3,7,10,11,17,18), (15,16,19,20). Our algorithm, however, generates five fragments in one

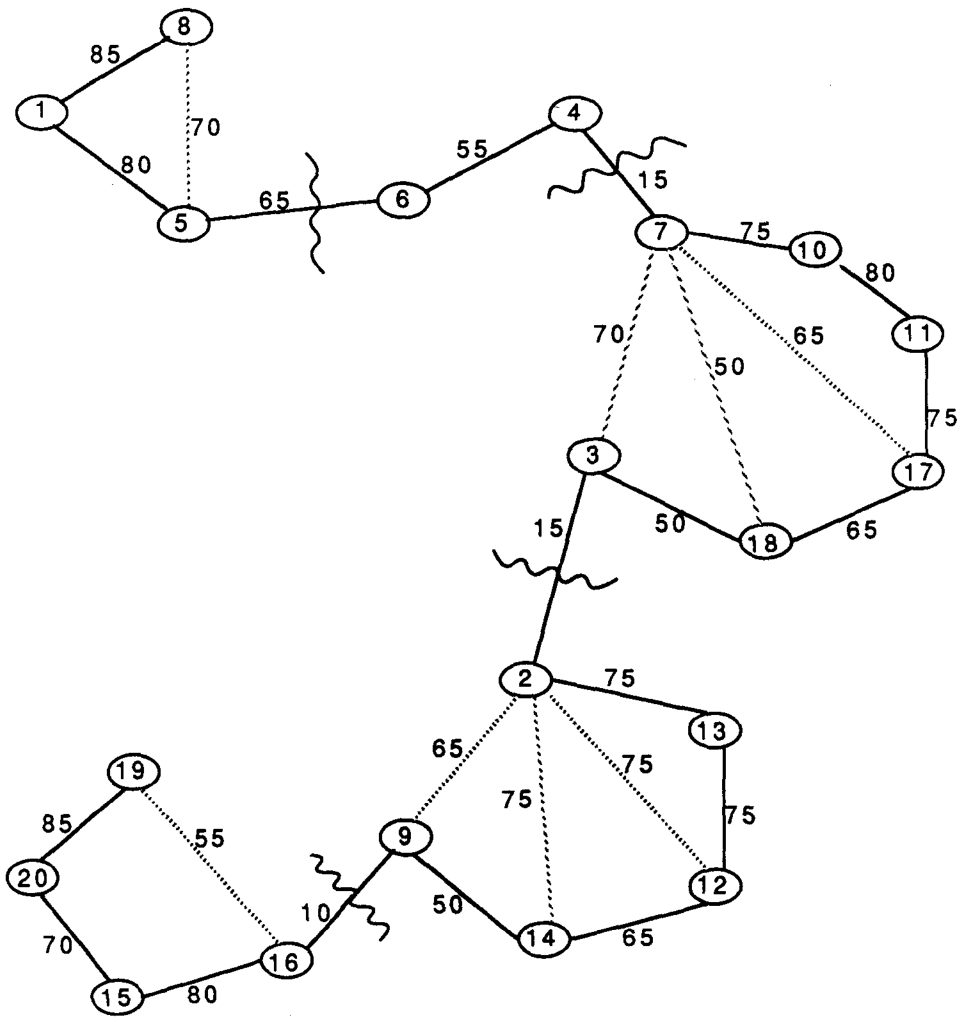

Fig. 8 Result of the second example

iteration as shown in Figure 8: (1,5,8), (4,6), (2,9,12,13,14), (3,7,10,11,17,18), (15,16,19,20). Note that the algorithm starts from node 1 and the cut of edge 3-2 is performed earlier than that of edge 4-7. This result shows that our algorithm can find one more possibility of partitioning. Thus what the empirical objective function could not discriminate as a potential partitioning in [Nava 84], is actually detected by our procedure.

3.4 Complexity of the algorithm

Now we consider the computational complexity. Step 1 does not affect the computational complexity because the attribute affinity matrix can be used as a symmetric matrix L. The *repeat* loop in the detailed description is executed n-1 times, where n denotes the number of attributes. At each iteration, selection of the next edge takes a time O(n). Also whether a cycle exists or not can be implemented in time of O(n) by scanning the vector B. Thus the algorithm takes a time O(n^2), which is less than that of [Nava 84], namely, O(n^2 log n).

4. Application

This algorithm can be used effectively for vertical partitioning because it overcomes the shortcomings of binary partitioning and it does not need any complementary algorithms such as the SHIFT and CLUSTER procedures that are used in [Nava 84]. Furthermore, the algorithm involves no arbitrary empirical objective functions to evaluate candidate partitions such as those used in [Nava 84].

Also this algorithm can be used for the cost-optimized vertical partitioning approach including different memory level partitioning and multiple site partitioning [Nava 84]. This application is currently being researched. We think it can be achieved by adding and analyzing an additional graph which contains cost information.

Another important application of this algorithm is the *mixed partitioning tool* which is under development in our D^3T project. The mixed partitioning tool that we are currently researching will first generate a grid for a relation vertically and horizontally, and then merge cells as much as possible by using a cost function for determining a fragment. We propose to implement the present algorithm for generating the grid vertically because it can generate all fragments of a relation simultaneously in one iteration.

The application of the proposed algorithm is in no way limited to just database design. The problem of clustering the nodes of a graph on the basis of affinity among the nodes can represent a variety of real life problems. They range from domains such as network design in communication to questionnaire design in social sciences where early work on clustering was done (e.g. see McCo 72). We see a vast potential for applying the proposed technique to a variety of domains.

5. Conclusion

We have described an algorithm for vertical partitioning, which uses a graphical technique. The major feature of this algorithm is that all fragments are generated by one iteration in a time of $O(n^2)$ that is more efficient than the previous approaches. Furthermore, it does not need any arbitrary objective function.

This algorithm can be further enhanced to address the problem of primary/secondary memory partitioning, or in the context of any memory hierarchy. By combining with the previously proposed MULTI_ALLOCATE algorithm, this algorithm can be used to achieve the allocation of vertical fragments over a network. Potential application of this algorithm can be in any domain where clustering on the basis of affinity is possible and meaningful.

Further extension of this research will be in the direction of developing an interactive design tool. This design tool will allow users to make fragmentation and allocation decisions for distributed databases using vertical, horizontal, and mixed partitioning. The present algorithm is being implemented for generating the vertical partitioning candidates in this mixed partitioning scenario. Extension of the present work to incorporate cost information will also be undertaken.

Acknowledgements

We thank Arunkumar Balaraman, Kamalakar Karlapalem, Rajeev Kumar, Pedro Rivera-Vega, and Prof. Ravi Varadarajan of the Distributed Database Design Tool (D^3T) project at the University of Florida for their constructive criticism of this approach. The research is supported by NSF Grant No. IRI 8716798.

References

[Bras 88] Brassard, G., and Bratley, P., *Algorithmics: Theory & Practice*, Prentice Hall, 1988.

[Ceri 84] Ceri, S., and Pelagatti, G., *Distributed Databases: Principles and Systems*, McGraw Hill, 1984.

[Ceri 88] Ceri, S., Pernici, B., and Wiederhold, G., "Optimization Problems and Solution Methods in the Design of Data Distribution," Working Paper, Stanford University, 1988.

[Corn 87] Cornell, D., and Yu, P. S., "A Vertical Partitioning Algorithm for Relational Databases," *Proc. Third International Conference on Data Engineering*, Feb. 1987.

[Hoff 75] Hoffer, J. A., and Severance, D. G., "The Use of Cluster Analysis in Physical Database Design," *Proc. First International Conference on Very Large Data Bases*, 1975.

[McCo 72] McCormick, W. T., Schweitzer, P. J., and White, T. W., "Problem Decomposition and Data Reorganization by a Clustering Technique," *Operations Research*, 20, Sep. 1972.

[Nava 84] Navathe, S. B., Ceri, S., Wiederhold, G., and Dou, J., "Vertical Partitioning Algorithms for Database Design," *ACM Trans. on Database Systems*, Vol. 9, No. 4, Dec. 1984.

[Nava 85] Navathe, S. B., and Ceri, S., "A Comprehensive Approach to Fragmentation and Allocation of Data in Distributed Databases," *IEEE Tutorial on Distributed Database Management*, (Larson,J.A., and Rahimi,S., eds), 1985.