VIEW INDEXING

IN RELATIONAL DATABASES

Nicholas Roussopoulos*

Larry S. Davis*

TR-101 June, 1979

*Computer Sciences Department, The University of Texas at Austin, Austin, Texas 78712

Reproduction of this report was paid by HITACHI.

ABSTRACT

The design and development of a useful database system requires that the physical organization of the database usefully reflects its logical organization . Views (classes of queries given by a Query model) are an appropriate intermediate logical representation for many databases. Frequently accessed views of databases need to be supported by indexing to enhance retrieval time. This paper investigates the problem of selecting an optimal index set of views and describes an efficient algorithm for this selection. The introduced redundancy in the database is controlled by an upper-bound storage constraint which is an input parameter to the algorithm.

1. INTRODUCTION

Indexing is a valuable technique for enhancing retrieval efficiency of large databases. There are many variations of indexing organizations, but they all basically allow searching to be performed via certain levels of indirection which drastically reduce the search effort.

Indexing in a database introduces overhead because it requires additional storage and processing to maintain the index sets during updates. However, in many applications the retrieval efficiency obtained through indexing more than offsets the storage and maintenance overhead.

Most previous research in database indexing concentrated on the selection of secondary indexes for a single file [AND77,CAR75,FAR75,KIN74,MAR78,SCH75, SCH78]. A secondary index on an attribute of a particular file is a collection of linked lists each of which links all of the records from that file which have the same value for that attribute. All the above studies assumed that accesses to the databases were described by a query model. The query model ordinarily consists of a set of ordered pairs, $\{(q_i,p_i)\}$, where p_i is the probability that query q_i will be made against the database. Then, an optimization objective function is developed which reflects the cost of answering all queries involving a particular file given a specific indexing of the file. Finally, algorithms are described which select a subset of the attributes of the file to be indexed which optimize the objective function.

In this paper we investigate the problem of selecting an optimal index set for efficient retrieval of views. Views are files (or relations) which are obtained from the original base relations using the relational algebra. Views have been incorporated in relational database systems such as

ZETA [MYL75], System R [AST76] and INGRESS [STO76].

A system which supports the creation of views can either

1) store the definition of each view, (i.e. the sequence of operators used to construct it) or,

2) store an index whose elements point to the tuples of the base relations which comprise the view, or

3) explicitly store the tuples of the view.

This last option is impractical because of the high cost of storage and maintenance. Clearly, retrieval of a view using the index set would be much more efficient with respect to both I/O operations and CPU time than executing the definition of the view. However, the cost of maintaining an index for each view may be very high with respect to storage and CPU needed to update the indexes whenever a change in the base relations is made.

Schkolnick [SCH78] characterized the cost of answering queries in a database system as a function of the resources available to that system. He points out that different cost functions should be developed for systems that are either

a) secondary storage bound,

b) I/O bound, or

c) CPU bound.

Schkolnick notes that I/O and CPU are highly interdependent since a substantial CPU load is due to I/O handling.

In dealing with views we note that all three cost functions are highly interdependent. I/O and CPU costs are high for a view if no index set is maintained since large parts of the database must be read and then processed using the view definition to produce the view. On the other hand, in a large database system with a large number of views, the overall secondary

storage and maintenance cost associated with indexes is very high.

The cost of answering a query against an indexed view is estimated in terms of I/O operations needed to retrieve the actual data. If the view is not indexed, the construction cost of the index is added. This construction cost depends on the size of the operands appearing in the definition of the view as well as on the complexity of computing the operations which define the view (e.g., the join operator has higher construction cost than the restriction or the projection operators).

We now define an optimization problem associated with view retrieval and storage. We assume that we are given

1) a set of views

2) the sizes of the indexes for those views,

3) the construction cost of those indexes, and

4) the probabilities of making queries against those views.

We would like to find the subset of these views which, when indexed, minimizes the total cost of answering all queries in the query model. Clearly, the choice of which indexes to store explicitly depends on the amount of secondary storage available - i.e., the storage constraint. If the storage is large enough to accommodate the indexes of all views, then the optimal subset would be the set of all views. If, on the other hand, the available secondary storage excludes any index to be stored explicitly, then the optimal subset is the empty set.

The above optimization problem does not deal directly with the cost of maintaining view indexes during updates. However, this maintenance cost is implicitly dealt with by the overall secondary storage constraint. That is, the maintenance cost of indexes can be adjusted by tuning the redundancy

implied by the storage constraint.

There are 2^N subsets of views, where N is the total number of views considered. Therefore a simple enumeration algorithm which computes the total cost of answering all queries in each subset is not a practical solution to the problem. In fact, the above problem is equivalent to the class of equivalent problems (NP-complete) for which no known polynomial algorithm exists.

In this paper we present an efficient algorithm which selects a set of indexes that satisfy the overall storage constraint while maximizing the optimization criteria. Although the algorithm has exponential worst case complexity (e.g., when all costs of index construction and sizes are identical), in practice we have found that only a very small percentage of the 2^N possible subsets are examined.

The query model, which we will assume is given, contains queries which are obtained through any relationally complete language. A complex query may generate intermediate relations which could be views for other queries. Therefore a "composition" of all these views is necessary prior to the application of the index selection algorithm. Chang & Cheng in [CHA78] have considered similar organization of databases although they considered only two relational operators: restriction and projection.

The remainder of the paper is organized as follows. Section 2 contains the problem formulation. This section assumes that all views that are considered for indexing have been composed and given to the algorithm in a graph notation. Section 3 describes the algorithm and proves that it is "admissible", i.e., that it is guaranteed to find the optimal solution. Section 4 describes the rules for obtaining the graph composition of views created by the query model. Section 5 describes some experimental results and, finally Section 6 contains conclusions.

2. PROBLEM FORMULATION

In this section we will describe a general form of the optimization problem outlined in the previous section. Here, we assume that the database is organized in a specific graph structure called a directed, acyclic AND/OR graph (DAAOG). A directed AND/OR graph is defined as follows:

Definition: $G = < N, E, D >$ is a directed AND/OR graph (DAOG) where:

1) $N = \{n_1, n_2, \ldots, n_m\}$ is a set of nodes
2) $E \subseteq N \times N$ is a set of directed edges, and
3) let $a(n_i) = \{n_j : (n_j, n_i) \in E\}$. Then $D: n_i \rightarrow 2^{a(n_i)}$ is called the decomposition mapping for the DAOG.

An acyclic DAOG (DAAOG) is a DAOG with no cycles.

Intuitively, each node in the DAAOG is associated with a relation, R_i. If $a(n_i) = \emptyset$, then R_i is one of the original relations or base relations in the database. Otherwise, R_i is called a derived relation. If R_i is a derived relation, and if $\ell \in D(n_i)$, then R_i can be constructed using suitable operations (union, join, etc. - see section 4) from the relations associated with the nodes in ℓ. $D(n_i)$ denotes the set of all possible sets ℓ's from which view n_i can be contructed.

Each relation, R_i, has a size, $|R_i|$, which corresponds to the amount of storage required to store R_i. In practice, as discussed in Section 1, R is not stored explicitly, but is represented by an appropriate index set into the base relations.

Each node, n_i, also has a probability, p_i, associated with it. This is the probability that a query entering the database system will be answered

based on the information in R_i.

Our goal is to choose a subset of the R_i to store explicitly which minimizes the cost of answering queries while satisfying a secondary storage constraint. Now, if there were no constraint on storage capacity, then independent of our cost model for answering a query, we could do no better than storing all the R_i. However, if there is a storage capacity constraint, i.e., if available storage $S < \sum_{i=1}^{m} |R_i|$ then certain relations cannot be stored explicitly, but must be constructed as needed. In order to choose which R_i to store explicitly, we will need a specific cost model for answering a query. In order to define our cost model, we first need the following definitions:

Definition: An allocation function, A, over a DAAOG, < N, E, D > is a mapping $A: N \rightarrow \{0,1\}$.

Definition: A construction cost function, cons, over a DAAOG, < N, E, D > is a mapping

$$\text{cons}: (\ell, n_i) \rightarrow [0,\infty) \quad \text{, where } \ell \in D(n_i).$$

A construction cost function measures the cost of constructing R_i from the sets in $D(n_i)$.

Next, we define the cost of answering a query at node n_i given the allocation, A, and the construction cost function, cons, as

$$C(n_i) = \begin{cases} c_r(R_i), \text{ if } A(n_i) = 1 \\ \min\limits_{\ell \in D(n_i)} [\text{ cons } (\ell, n_i) + \sum\limits_{n_j \in \ell} C(n_j)], \text{ if } A(n_i) = 0 \end{cases}$$

C is designed to reflect the cost, in I/O operations, of answering a query at n_i. The function c_r reflects the cost in I/O operations of acquiring R_i. If R_i is stored explicitly, then c_r is a function linear in $|R_i|$. If cons (ℓ, n_i) reflects the number of I/O operations required to construct R_i from ℓ, then cons $(\ell, n_i) + \sum_{n_j \in \ell} C(n_j)$ obviously reflects the number of I/O operations required to first construct ℓ and then R_i. Clearly, C can be changed to reflect, e.g., a measure of CPU cost for answering a query.

Finally, we define an optimal allocation, A, for a DAAOG, < N, E, D > and a storage constraint, S, to be any A such that:

1) $\sum_{i=1}^{m} C(n_i)\, p_i$ is minimal, and

2) $\sum_{i=1}^{m} A(n_i)\, |R_i| \leq S$

Our optimality criteria does not explicitly take into account the cost of updating a database organized as a DAAOG. There are two reasons for this omission:

1) updates can conceivably cause gross changes in the organization of the database, and are therefore difficult to model, and

2) if the R_i are stored as index sets, then updates require few operations relative to the costly retrieval of the index sets.

Notice that we could have reversed the optimization criterion - i.e., choose A to make $\sum A(n_i)|R_i|$ minimal while keeping $\sum C(n_i)\, p_i$ below some threshold. Or, we could have modified the criteria to make $\sum A(n_i)\, |R_i|$

minimal while keeping $C(n_i)$ below threshold for all i. This would guarantee that the cost of answering any query will never exceed the given threshold. This optimization criterion would make the cost of answering queries more uniform than the two previous criteria.

The optimal allocation problem is equivalent to the data allocation problem and has been shown [CHA77] to be difficult even when some simplifying assumptions are made (such as assuming all relations have equal size). It belongs to a class of equally difficult problems called NP-complete problems and no polynomial algorithm is known for this class. In the following section we describe a solution to the problem of constructing an optimal allocation for a database. The solution is based on the A* algorithm described in [NIL71]. Experimental results (Section 5) indicate that the algorithm performs much better than its worst case analysis would suggest.

3. COMPUTING OPTIMAL ALLOCATIONS

In this section we will describe the procedure for computing the optimal indexing allocation function A. The procedure is based on the A* algorithm (see [NIL71]) and uses a fast approximate algorithm for the Knapsack problem to compute a heuristic function (see [SAH75] for a description of the Knapsack algorithm).

Let $N = \{n_1, n_2, \ldots\ldots, n_m\}$ be the nodes of the DAAOG, $< N,E,D >$. Let $r(n_i)$ be the size of the index set for the relation associated with node n_i. Let $C(n_i)$ denote the cost of answering a query using the index set at n_i. Let $p(n_i)$ be the probability that a query entering the system will be answered using the index set stored at n_i. Finally, suppose S is a global storage constraint governing the total size of the index sets which may be stored explicitly.

Then the goal is to find the allocation A such that

$$\sum_{i=1}^{m} C(n_i)p(n_i)$$

is minimal, while guaranteeing <u>feasibility</u>, i.e.,

$$\sum_{i=1}^{m} r(n_i)A(n_i) \leq S$$

We will define a state-space representation for this problem, and then define an ordering function of the form $\hat{f} = g+\hat{h}$ which is guaranteed to guide search towards the optimal solution. We should point out that the optimization problem can also be formulated as a dynamic programming (DP)

problem. However, the storage requirements of a DP solution to our problem are prohibitive.

We define a state, s, to be a 5-tuple

$$\langle N_s, A_s, S_s, g_s, \hat{h}_s \rangle$$

where

1) $N_s \subseteq N$ subject to the constraint that if $n_i \in N_s$, then $a(n_i) \subseteq N_s$.

Thus, when we add a node to N_s to produce a new state, s', (see below), we can compute the actual cost (contingent on A_s) of answering a query based on the index set at n_i or the index set of its ancestors $a(n_i)$.

2) $A_s: N_s \rightarrow \{0,1\}$ is the allocation function for state s. Clearly

$$\sum_{n \in N_s} r(n) A_s(n) \leq S$$

3) $S_s = S - \sum_{n \in N_s} A_s(n) r(n)$. S_s is the storage left at state s available to be allocated to the remaining $N-N_s$ nodes

4) $g_s = \sum_{n \in N_s} C_s(n) p(n)$. $C_s(n)$ is the recursive cost function as defined in Section 2. The subscript s indicates that it is contingent on A_s; g_s reflects the total actual cost of answering queries involving only those nodes in N_s.

5) $\hat{h}_s$ is an estimate of the incremental cost that must be incurred in extending A_s from an allocation on N_s to an allocation on N.

$$\hat{h}_s = k \sum_{n \in N-N_s} p(n)\ \bar{\delta}(n)\ \min_{\ell \in D(n)} [\text{cons}\ (\ell, n)]$$

where $\bar{\delta}$ is the complement of another allocation function δ:

$$\delta: N-N_s \rightarrow \{0,1\}$$

computed by the knapsack algorithm (see below) and k is a constant. If h_s is the minimal cost of all possible extensions of A_s to A_N, then we will show that

$$\hat{h}_s < h_s.$$

It is this lower bound property that guarantees that search ordered by the evaluation function $\hat{f}_s = g_s + \hat{h}_s$ will terminate with the optimal allocation A_N (see [NIL71]).

The start state is

$$<N_R, A_R, S_R, g_R, \hat{h}_R>$$

where

1) N_R is the set of nodes associated with the base relations

2) $A_R : N_R \rightarrow \{1\}$. This guarantees that all base relations are stored.

3) $S_R = S - \sum_{n \in N_R} r(n)$. If $S_R > S$, there are no feasible solutions. If $S_R = S$, no redundancy is allowed. In both situations the algorithm stops immediately.

4) $g_R = \sum_{n \in N_R} C(n)p(n)$

5) $\hat{h}_R = k \sum_{n \in N-N_R} p(n)\,\bar{\delta}(n) \min_{\ell \in D(n)}[cons(\ell,n)]$

A final state is of the form $< N, A_N, S_N, g, 0 >$

Given a state, s, we generate its successors s' by

1) choosing an $n_i \in N-N_s$ such that $a(n_i) \subseteq N_s$. In other words n_i is chosen only if its ancestors have already been considered.

2) Let n_i be such a node. We will create at least one and possibly two new states. There will always be the new state with $A_{s'}(n_i) = 0$.

If

$$S - \sum_{n \in N_s} r(n) \geq r(n_i)$$

then we can also create the state s" with $A_{s"}(n_i) = 1$.

Search proceeds by first putting the start state on a list called OPEN. At each stage of the search, we remove from OPEN the state s with minimal $\hat{f}_s = g_s + \hat{h}_s$. If s is a final state, search halts. Otherwise, we generate the successors at s and place them on OPEN.

In order to prove that the first final state removed from OPEN represents the optimal solution, we must show $\hat{h}_s \leq h_s$. We shall first describe our heuristic function.

Let $s = < N_s, A_s, S_s, g_s, \hat{h}_s >$ be a state. We use an approximate solution to the knapsack problem to make a good guess as to which nodes in $N-N_s$ will receive the remaining storage S_s. In the knapsack problem we are given a set of ordered pairs, $E = \left\{(s_1,p_1), (s_2,p_2),\ldots,(s_t,p_t)\right\}$, of storage costs (s_i) and profits (p_i), and a knapsack of capacity G. We want to find the subset $E' \subseteq E$ such that

$$P_{E'} = \sum_{(s_i,p_i) \in E'} p_i$$

is maximal, subject to the constraint that

$$\sum_{(s_i,p_i) \in E'} s_i \leq G.$$

An alternative statement of the problem is that we want to find an allocation, δ,

$\delta: E \rightarrow \{0,1\}$ such that

$$\sum_{(s_i,p_i) \in E} p_i \, \delta((s_i,p_i))$$

is maximal, subject to the above constraint.

It is well known that the knapsack problem is in the class of NP-complete problems [KAR72]. However, fast approximate solutions have been described (see, e.g. [SAH75]).

For our problem $G = S_s$, and $E = \{(s_i,p_i)$: $s_i = r(n_i)$, $n_i \in N-N_s$ and

$$p_i = \min_{\ell \in D(n)} [cons(\ell,n)]$$

The approximate algorithm for the knapsack problem computes a set E" such that

$$P(E') < (1 + \frac{1}{k})\ P(E'') \tag{1}$$

where k is a positive integer input parameter to the algorithm. The larger the value of k the larger the computational requirements of the algorithm and the better the approximation of P(E") to P(E').

Let δ: $E \rightarrow \{0,1\}$ be defined as follows

$$\delta(n_i) = \begin{cases} 1 \text{ if } (s_i,p_i) \in E'' \\ 0 \text{ otherwise} \end{cases}$$

we then define

$$\hat{h}_s = (1 + \frac{1}{k}) \sum_{n \in N-N_s} p(n)\ \bar{\delta}\ (n)\ \min_{\ell \in D(n)} [cons(\ell,n)]$$

where $\bar{\delta}$ is the complement of δ.

Proposition: $\hat{h}_s \leq h_s$

Proof: Let δ_{h_s} be the optimal allocation which best extends A_s to A_N and

$$h_s = \sum_{n_i \in N-N_s} p(n_i)\ C(n_i)$$

be the minimal incremental cost that is incurred by δ_{h_s}.

Let

$$M_s = \sum_{n_i \in N-N_s} p(n_i)\ \bar{\delta}_{b_s}(n_i)\ \min_{\ell \in D(n_i)} [cons\ (\ell,n_i)]$$

where

$$\delta_{b_s}(n_i) = \begin{cases} 1 & \text{if } n_i \text{ corresponds to a base relation} \\ 0 & \text{otherwise} \end{cases}$$

Let also

$$P(E^*) = \sum_{n_i \in N-N_s} p(n_i)\ \bar{\delta}_{h_s}(n_i) \quad \min_{n_i \in D(n_i)}[cons(\ell, n_i)]$$

Then clearly

$$h_s \geq M_s - P(E^*) \tag{2}$$

because $P(E^*)$ is only the sum of the costs of constructing the non-allocated views by δ_{h_s} without any consideration to the recursive cost. But

$$M_s - P(E^*) \geq M_s - P(E') \tag{3}$$

because E^* and E' are both feasible solutions to the knapsack problem and E' is the optimal one.

From equations (1) and (3) we obtain

$$M_s - P(E') \geq M_s - (1+\frac{1}{k})P(E'') = \hat{h}_s \tag{4}$$

and from (2), (3) and (4)

$$h_s \geq M_s - P(E^*) \geq M_s - P(E') \geq M_s - (1+\frac{1}{k})\ P(E'') = \hat{h}_s.$$

It should be pointed out that the constant $1+\frac{1}{k}$ can be replaced by

$$\min\{\ 1+\frac{1}{k},\ \frac{(P(E''))^2}{P(E'')-\bar{p}}\ \}$$

where $\bar{p}$ is the k+1-st largest p_i passed to the knapsack algorithm [SAH75], without affecting the proof of the above proposition. In most cases $\frac{(P(E''))^2}{P(E'')-\bar{p}}$ is a better lower bound and thus the resulting $\hat{h}_s$ a better approximation of h_s.

4. CREATING DAAOG REPRESENTATIONS FOR RELATIONAL DATA BASES

In this section we will briefly describe one way to organize a Relational Database into a DAAOG. A more detailed discussion is available in Roussopoulos and Davis [ROU79].

We assume that the database designer has a set of relations (or files), $R = \{R_i\}_{i=1}^{n}$ which will form the database. In addition, he has a query model, $QM = \{(q_i,p_i)\}_{i=1}^{m}$. Here, q_i is a query and p_i is the probability that this query will be directed at the system. Thus $\sum_{i=1}^{m} p_i = 1$. The q_i will ordinarily represent a class of general queries rather than a query concerning specific values of relational attributes - e.g., q_i might represent any query involving male, graduate students. QM may have been obtained from the prior requirements of the system, or from observations of an existing database system over a period of time.

Each q_i will be represented by what we call a query graph which is composed of views. We will first define what is meant by a view, and then present query graphs for several queries.

Definition 1: A view is defined recursively as follows:

1) If $R_i \in R$, then R_i is a view

2) Let V_i, V_j be two views. Then

a. $h_p(V_i)$ is a view, where h_p is called the horizontal selector or restriction operator; $h_p(V_i)$ is the subset of V_i which satisfies predicate P. For Example, if V_i is the set of students, then P might be the predicate GRADUATE. We adopt the graphical notation in Figure 1a to represent a

horizontal selector. The construction cost, cons, of obtaining $h_p(V_i)$ is a function linear in the size of V_i.

b. $v_A(V_i)$ is a view, where v_A is the vertical selector or projection operator. A is a subset of the attributes of V_i, and $v_A(V_i) = \pi_A V_i$, where π is the standard projection operator. Figure 1b contains the graphical notation of v_A. As with h_p, cons (v_A) is linear in the size of V_i.

c. $j_A(V_i,V_j)$ is a view, where j_A is the join operator. Here, A is again a subset of the attributes of V_i and V_j. Figure 1c shows the graphical notation for the join operator; cons (j_A) is linear in the product of the sizes of V_i and V_j.

d. $u(V_i,V_j)$ is a view, where u is the union operator; $u(V_i,V_j) = V_i \sqcup V_j$. Figure 1d shows the graphical notation for union; cons (u) is linear in the sum of the sizes of V_i and V_j.

e. $i(V_i,V_j)$ is a view, where i is the intersection operator, $i(V_i,V_j) = V_i \sqcap V_j$. Figure 1e shows the graphical notation for intersection; cons(i) is again, linear in the sum of the sizes of V_i and V_j.

f. $d(V_i,V_j)$ is a view, where v is the difference operator; $d(V_i,V_j) = V_i - V_j$. Figure 1f shows the graphical notation for difference; cons(d) is also linear in the sum of the sizes of V_i and V_j.

g. there are no other views.

Any query can be represented by a query graph because the set of the operators is relationally complete. A node in the query graph is a view, and views are related to one another through the operators defined above. As an example, Figure 2 contains a query graph for queries involving graduate student employees. Here, both STUDENT and EMPLOYEE are elements of R. Notice that corresponding to any query there are an infinite

number of query graphs (since, for example, projections and join are inverse operators when the projection is determined by a functional dependency). We will assume that the data base designer constructs a single query graph for each query in the query model.

Attached to each node in the query for q_i will be p_i, the probability of that query. The p_i's are used as described in Section 3 to decide which of the views in the query graph will be stored explicitly in the database system, and which will be constructed as needed.

Two query graphs can be merged to form a single graph. We will describe the merging process through examples; a rigorous description is possible, but it is not included here (see [ROU79] for details). We will first consider merging query graphs containing the operators h_R,i,u and d only. Successive merges of query graphs results in an organization of the underlying views which has been called an ISA hierarchy [ROU76].

Figure 3a contains query graphs for the queries (q_1,p_1) and (q_2,p_2) where

1) q_1: MALE-STUDENT, and

2) q_2: GRADUATE-STUDENT

Figure 3b contains the merged graphs. Notice that since it is <u>not</u> the case that either

MALE-STUDENT(X) → GRADUATE-STUDENT(X), or

GRADUATE-STUDENT(X) → MALE-STUDENT(X)

the depth of the graph in Figure 3b is 1. If q_2 had been MALE-GRADUATE-STUDENT then the result of the merge would be as in Figure 3c. Note that if there are identical nodes in the graphs which are being merged, (such as the STUDENT nodes in Figure 3a), then the probability of the corresponding

node in the merged graph is the sum of the probabilities of the nodes in the original graphs.

Suppose, now, that we add the query (q_3,p_3) where q_3 is EMPLOYED-GRADUATE-STUDENTS. Its query graph is shown in Figure 4a. We can merge it with Figure 3b to obtain Figure 4b. Finally, if query (q_4,p_4) with q_4 being EMPLOYED-FEMALE-GRADUATE-STUDENTS is added (with query graph as shown in Figure 5a), then the final merged structures (ignoring the broken lines) is shown in Figure 5b.

After all of the query graphs have been merged, the data base designer may notice that simple alternatives might be available for constructing some of the views. So, for example, EMPLOYED-FEMALE-GRAUDATE-STUDENT can also be constructed by computing the intersection of FEMALE-GRADUATE STUDENT and EMPLOYED-GRADUATE-STUDENT. The dotted arc in Figure 5b can be added to represent this alternative. Note that if the designer does add this arc, then the probability of EMPLOYED-GRADUATE-STUDENT is increased by p_4. We could, conceivably, compute <u>all</u> alternative ways to construct any view from other views, but this computation would be quite expensive; instead the designer is relied on to add alternatives.

The result of the merging operations is a partial ordering very similar to the ISA hierarchy [ROU76]. The only distinction is that some transitive arcs are explicitly represented in the merged graph, while in the ISA-hierarchy, which is represented by a Hasse diagram, they would be implicit.

The second stage of the merge operation involves the projection and join operators. Consider the queries (q_5,p_5), (q_6,p_6) where:

1) q_5: GRADUATE-STUDENT-GRADE-RECORDS, and

2) q_6: MALE-GRADUATE-STUDENT-GRADE-RECORDS

Figure 6a-b contains query graphs for those queries, while Figure 6c contains the merged graph. In general, when dealing with join and projection operators, query graphs are merged as shown in Figure 7. Here V(A, B, C,) represents a view based on attributes A B and C.

Once all the query graphs have been merged, the result is a structure which can be represented as a DAAOG, which can in turn be analyzed based on the approach presented in Sections 2-3.

5. IMPLEMENTATION OF THE ALGORITHM

The algorithm described in this paper was implemented in PASCAL and runs on a CDC-6600 computer here at the University of Texas. Two significant modifications were made to improve computation time and reduce the size of the OPEN list.

The first was made after noticing that when the program generates a pair of states s' and s" one that corresponds to $A_{s'}(n_i) = 0$ and the other to $A_{s''}(n_i) = 1$ (see section 3), it is only necessary to call the approximate knapsack algorithm once. This can be done by looking at the result of the call of the ancestor state s of s' or s" in the OPEN list. Suppose that $A_s(n_{i-1}) = 0$ and the call KNAP(S_s,E_s), with $E_s=\{(s_j,p_j),\ i\leq j\leq m\}$, produced a $\bar{\delta}$ from which we computed $\hat{h}_s$. Now at state s', we know that if $\bar{\delta}_s(n_i)=0$, then the call KNAP($S_{s'}=S_s$,$E_{s'}$) with $E_{s'}=\{(s_j,p_j),\ i+1\leq j\leq m\}$, would produce a $\bar{\delta}_{s'}$, which is equal to $\bar{\delta}_s$ in all places but the i-th, (i.e. $\bar{\delta}_{s'}(n_j)=\bar{\delta}_s(n_j)$ for $i+1\leq j\leq m$), because $S_{s'}=S_s$, $E_{s'}=E_s-\{(s_i,p_i)\}$ and (s_i,p_i) was not put in the knapsack at the first call of the algorithm. Thus, we can compute the heuristic $\hat{h}_{s'}$ without calling again the approximate knapsack program. If, on the other hand, $\bar{\delta}_s(n_i)$ was 1, then the call has to be made.

The other case is for the state s", i.e. $A_{s''}(n_i)=1$. The call KNAP($S_{s''}=S_s-r(n_i)$,$E_{s''}$) with $E_{s''}=\{(s_j,p_j),\ i+1\leq j\leq m\}$ needs not to be made if $\bar{\delta}_s(n_i)=1$. This is because the knapsack program would produce a $\bar{\delta}_{s''}$ which is equal to $\bar{\delta}_s$, i.e. $\bar{\delta}_{s''}(n_j)=\bar{\delta}_s(n_j)$, $i+1\leq j\leq m$. Therefore, again in this case, the heuristic function $\hat{h}_{s''}$ can be computed without a separate call. For $A_{s''}(n_i)=1$ and $\bar{\delta}_s(n_i)=0$ the call has to be made. This improvement is significant with respect to CPU time because half of the calls to the approximate knapsack algorithm are avoided.

The second modification was to improve the heuristic $\hat{h}_s$ and make it as close to h_s but still bound by h_s, i.e. $\hat{h}_s < h_s$, in order to guarantee its admissibility. It was noticed that the algorithm was behaving quite well in relatively large storage constraints and/or shallow DAOG's but not as well when the storage constraint was relatively small and/or the height of the DAOG's large. This is true because the contributed accumulating recursive cost is large and the described heuristic function was not taking this recursive cost into account. When the storage is large enough or in shallow trees, however, this accumulating recursive cost is very small.

To overcome this problem we added an additional factor to the heuristic function which accounts for some of the accumulating cost. Let $h_s = h_{s_1} + h_{s_2}$ be the optimal incremental cost at a state s. Here h_{s_1} corresponds to the cost contributed by the construction costs of the non-indexed unvisited nodes in an optimal allocation (assuming that their immediate parents are indexed). h_{s_2} is the recursive accumulated cost of the same allocation.
Also, let

$$M'_s = \sum_{n_i \in N-N_s} p(n_i)\, C(n_i)$$

Then we know that the accumlating cost of any allocation is less than or equal to

$$\Delta M_s = M'_s - M_s$$

where M_s is as defined in the proposition of section 3, i.e.

$$M_s = \sum_{n_i \in N-N_s} p(n_i)\, \overline{\delta}_{b_s}(n_i) \min_{\ell \in D(n_i)}[cons(\ell, n_i)]$$

with

$$\delta_{b_s}(n_i) = \begin{cases} 1 \text{ if } n_i \text{ corresponds to a base relation} \\ 0 \text{ otherwise} \end{cases}$$

Therefore $h_{s_2} < \Delta M_s$. ΔM_s is the maximum accumulating cost assuming that no unvisited node is indexed.

We now define the quantity RAC (Reduction of the Accumulating Cost) by

$$RAC(n_i) = M'_s - [\sum_{n_j \epsilon N-N_s} p(n_j)\ \bar{\delta}_{n_i}(n_j)\ C(n_j)] - p(n_i)\ \min_{\ell \epsilon D(n_i)}[cons(\ell, n_i)]$$

where

$$\delta_{n_i}(n_j) = \begin{cases} 1 & \text{if } n_j = n_i \text{ or } n_j \text{ corresponds to a base relation} \\ 0 & \text{otherwise.} \end{cases}$$

This quantity corresponds to the reduction of the accumulating recursive cost attained by allocating storage to node n_i only. Note that for any feasible allocation the sum of the corresponding RAC's of the allocated nodes is clearly larger than the reduction in actual accumulating cost. This is because the RAC's were computed as if they were independent of each other.

Using the RAC's, the storage requirements of the nodes, and the same storage constraint S_s at each state, the knapsack algorithm is run again. Let $P(E'_2)$ be the optimal solution to the knapsack problem using RAC's and let $P(E''_2)$ be the one that the approximate algorithm computes. Then again, we have

$$P(E'_2) \leq (1 + \frac{1}{k})P(E''_2) \qquad (1)$$

$P(E'_2)$ is a quantity which corresponds to an allocation that produces the largest possible reduction of the accumulating cost and therefore is larger than or equal to the reduction produced by the overall optimal allocation $P(E_2^*)$

$$P(E_2^*) \leq P(E'_2) \qquad (2)$$

But then from (1) and (2) we have

$$h_{2_s} = \Delta M_s - P(E_2^*) \geq \Delta M_s - P(E'_2) \geq \Delta M_s - (1 + \frac{1}{k})P(E''_2)$$

and thus if we let

$$\hat{h}_{s_2} = \min\{0,\ \Delta M_s - (1 + \frac{1}{k})P(E''_2)\}$$

we have

$$h_s = h_{s_1} + h_{s_2} \geq \hat{h}_{s_1} + \hat{h}_{s_2} = \hat{h}_s$$

The $\hat{h}_{s_2}$ component increases the heuristic function and speeds up the search especially at the early stages without destroying the admissibility of the algorithm.

It is important to note here that the RAC's and the Δm's are computed only once before the search starts. This saves a great deal of computation. Furthermore, when ΔM is 0 or has a small value, then the extra call to the knapsack is not made. This situation occurs in relatively flat DAOG's or during the late stages of the search when the structure of the remaining unvisited DAOG is flat.

In order to simplify the above proof of admissibility, we assumed that the local cost of retrieving an indexed view $c_r(n_i)$ is 0. However, the algorithm is still admissible even if we assume that there is a uniform distribution of these costs proportional to the view size.

We present below a number of examples and the results produced by the program. In order to make these tables interpretable we used the same probabilities for all views and assumed that the cost of answering a query at every indexed node is zero.

Each example was run using a number of different storage constraints starting with S_B, which is the storage required to store all base relations, and up to S_{ALL} which is all the storage it takes to index all views. The figures under the cost column refer to the cost of the optimal allocation, which is the cost to answer all queries. CPU time required to generate each solution is not given but can be inferred by the length of the OPEN list. It took about three minutes to generate 10,000 entries for the OPEN list.

Next to the OPEN list length is the ratio of this length over the total number of possible allocations. Finally, the last column is the optimal allocation for the given storage constraint. Both examples, shown here were created using the unary operators horizontal and vertical selector because these are easier to verify.

STORAGE CONSTRAINT	COST	OPEN LIST LENGTH	RATIO 0/00	NODE OPTIMAL ALLOCATION 1 2 3 4 5 6 7 8 9 10 11 12 13
S_B	657	26	3	1 0 0 0 0 0 0 0 0 0 0 0 0
1.2 S_B	577	66	8	1 0 0 0 0 0 0 0 0 0 0 1 0
1.4 S_B	357	66	8	1 0 0 1 0 0 0 0 0 0 0 0 0
1.6 S_B	305	136	16	1 0 0 1 0 0 0 0 0 0 1 0 0
1.8 S_B	234	120	14	1 0 0 1 0 0 1 0 0 0 0 0 0
2.0 S_B	184	82	10	1 0 0 1 1 0 1 0 0 0 0 0 0
2.2 S_B	164	144	17	1 0 0 1 1 0 1 0 0 0 0 1 0
2.4 S_B	140	162	19	1 0 0 1 1 0 1 1 0 0 0 1 1
2.6 S_B	121	196	23	1 0 0 1 1 1 1 0 0 0 0 1 0
2.8 S_B	97	134	16	1 0 0 1 1 1 1 1 0 0 0 1 1
3.0 S_B	86	176	21	1 0 0 1 1 1 1 1 0 0 1 1 1
3.4 S_B	67	312	38	1 1 0 1 0 1 1 1 0 0 0 1 1
3.8 S_B	46	320	39	1 1 0 1 1 1 1 1 0 0 1 1 1
4.2 S_B	28	176	21	1 1 1 1 1 1 0 1 0 1 1 1 1
S_{ALL}=4.85 S_B	0	26	3	1 1 1 1 1 1 1 1 1 1 1 1 1

Table for figure 8

STORAGE CONSTRAINT	COST	OPEN LIST LENGTH	RATIO 0/00	NODE OPTIMAL ALLOCATION 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
S_B	3330	42	0	1 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0
1.2 S_B	2715	638	0.3	1 1 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 1 0 0 0
1.4 S_B	2130	4132	1.9	1 1 0 0 0 0 1 1 0 0 0 0 0 0 1 0 0 1 1 1 0
1.6 S_B	1645	9024	4.3	1 1 0 0 0 1 1 1 0 0 0 0 1 0 1 0 0 1 1 1 0
1.8 S_B	1230	9420	4.5	1 1 0 0 0 1 1 1 0 0 1 0 1 0 1 0 1 1 1 0 0
2.0 S_B	910	6832	3.2	1 1 0 1 0 1 1 1 0 0 1 1 1 0 1 1 1 0 0 0 0
2.2 S_B	630	1830	0.8	1 1 0 1 0 1 1 1 0 0 1 1 1 1 1 1 1 1 0 0 0
2.4 S_B	500	2154	1.0	1 1 0 1 0 1 1 1 0 1 1 1 1 1 1 1 1 1 0 0 0
2.6 S_B	380	1874	0.8	1 1 1 1 0 1 1 1 0 0 1 1 1 1 1 1 1 1 1 1 0
2.8 S_B	250	772	0.3	1 1 1 1 0 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 0
3.0 S_B	130	266	0.1	1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1
S_{ALL}=3.3 S_B	0	42	0	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

Table for figure 9

6. CONCLUSIONS

The design and development of a useful database system requires that the physical representation of the database usefully reflects its logical organization. We suggest that views are an appropriate intermediate logical representation for many databases. This paper has shown how the physical organization of a database can be determined by a set of views and a query model, in an optimal way. There are several important issues, however, which this paper did not address, but which are currently under investigation:

1) If a database has many views, then the efficiency of the algorithms presented in Sections 3 and 4 can be enhanced by introducing a hierarchical organization over set of views, and then performing a staged optimization--i.e., first allocating storage to the views at the highest level in the hierarchy, and then subsequently distributing that storage at lower and lower levels. It is important, for a variety of reasons, that the structure of the hierarchy reflects the semantic organization of the database as well as its syntactic organization. By the "semantic" organization we mean a logical description of how the various data are related to one another (perhaps as reflected in the query model which indicates which sets of data are generally required to answer queries). By the syntactic organization, we refer to certain formal relations between parts of the database which can be defined independent of its semantics--e.g., subset relations.

2) Methods should be developed for mapping queries into operations on views to complete the link from the logical organization of the data base, through the views, to its physical organization. Once this is accomplished problems such as planning answer strategies can be treated at the level of views, which are relatively abstract but sufficiently simple so that both semantic and syntactic constraints can be brought to bear on the planning process.

REFERENCES

[AND 77]

Anderson, H. D., and Berra, P. B., "Minimum Cost Selection of Secondary Indexes", ACM Transactions on Database Systems, vol. 2, No. 1, March 1977, pp. 68-90.

[AST 76]

Astrahan, M. M., et al, "System R: Relational Approach to Database Management", ACM Transactions on Database Systems, vol. 1, No. 2, June 1976, pp. 97-137.

[CAR 75]

Cardenas, A. F., "Analysis and Performance of Inverted Database Structures", Comm. of ACM, vol. 18, No. 5, May 1975, pp. 253-263.

[CHA 77]

Chandy, K. M., "Models of Distributed Systems", Proc. of Third VLDB, Tokyo, Japan, 1977, pp. 105-120.

[CHA 78]

Chang, S. K., and Cheng, W. H., "Database Skeleton and Its Application to Logical Database Syntheisi", IEEE Transactions on Software Engineering, vol. SE-U, No. 1, January 1978, pp. 18-30.

[FAR 75]

Farley, J. H. G. and Schuster, S. A., "Query Execution and Index Selection for Relational Data Bases", TR CSRG-53, University of Toronto, March 1975.

[KAR 72]

Karp, R. M., "Reducibility Among Combinatorial Problems", in Complexity of Computer Computations, Eds. Miller, R. E., and Thatcher, J. W., Plenum Press 1972, pp. 85-104.

[KIN 74]

King, W. F., "On the Selection of Indices for a File", IBM Research Report RJ1341, San Jose, CA, 1974.

[MAR 78]

March, S. T., and Severance, D. G., "A Mathematical Modelling Approach to the Automatic Selection of Database Design", Proc. of SIGMOD 1978, Austin, TX.

[MYL 75]

Mylopoulos, J., Schuster, and Tsichritzis, D., "A Multi-Level Relational System", Proceedings of NCC, 1975.

[NIL 71]

Nilsson, N. J., "Problem-Solving Methods in Artificial Intelligence, McGraw-Hill 1971.

[ROU 79]

Roussopoulos, N., Davis, L., and Shih, D., "An integrated View Indexing System", in preparation.

[SAH 75]

Sahni, S., "Approximate Algorithms for the 0/1 Knapsack Problem", Journal of ACM, vol. 22, No. 1, January 1975, pp. 115-124.

[SCH 75]

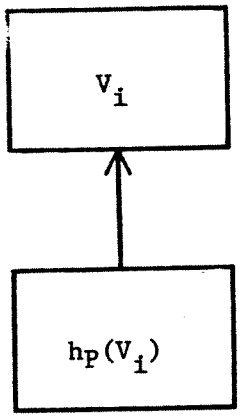
Schkolnick, M., "Secondary Index Optimization", Proc. ACM SIGMOD, 1975, pp. 186-192.

[SCH 78]

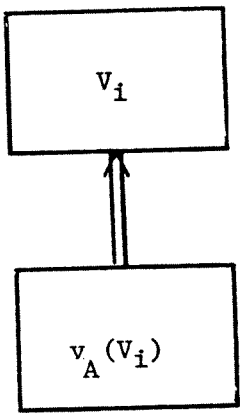
Schkolnick, M., "Physical Database Design Techniques", NYU Symposium on Database Design, New York 1978.

[STO 76]

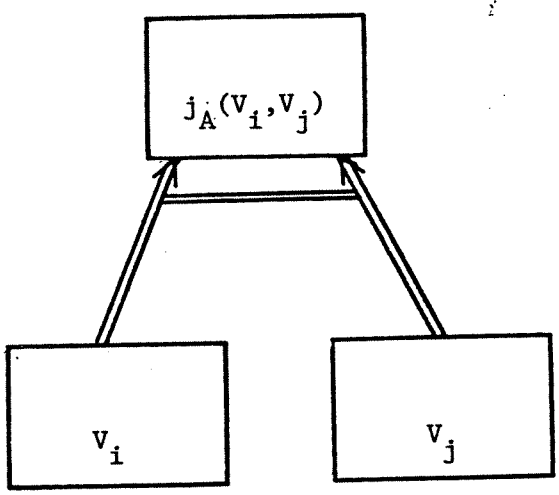
Stonebraker, M. R., Wong, E., Kreps, P., and Held, G., "The Design and Implementation of INGRESS", ACM Transactions on Database Systems, vol.1, No. 3, September 1976, pp. 189-222.



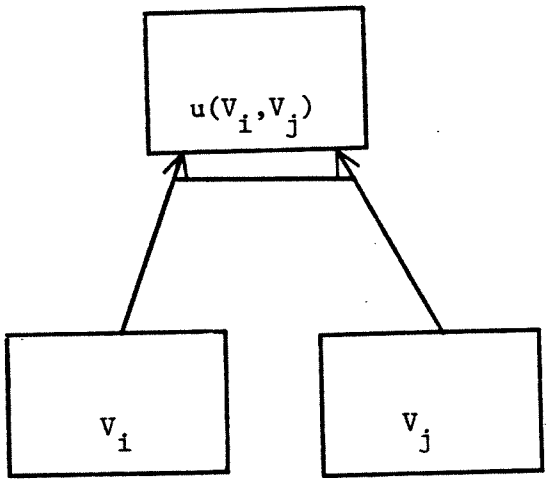
a) h_P



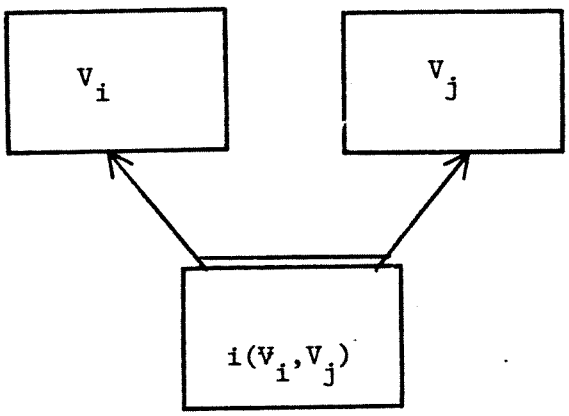
b) v_A



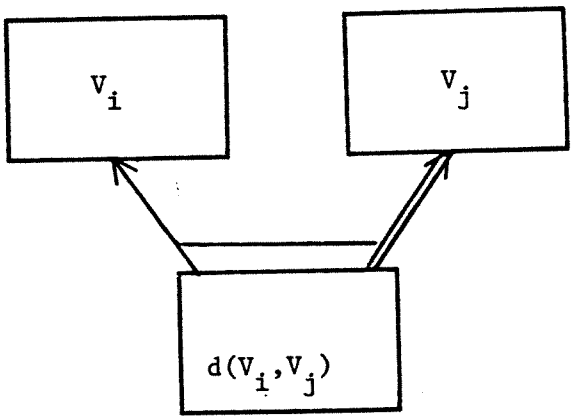
c) j_A



d) u



e) i



f) d

Figure 1. Graphical notation for the six relational operators.

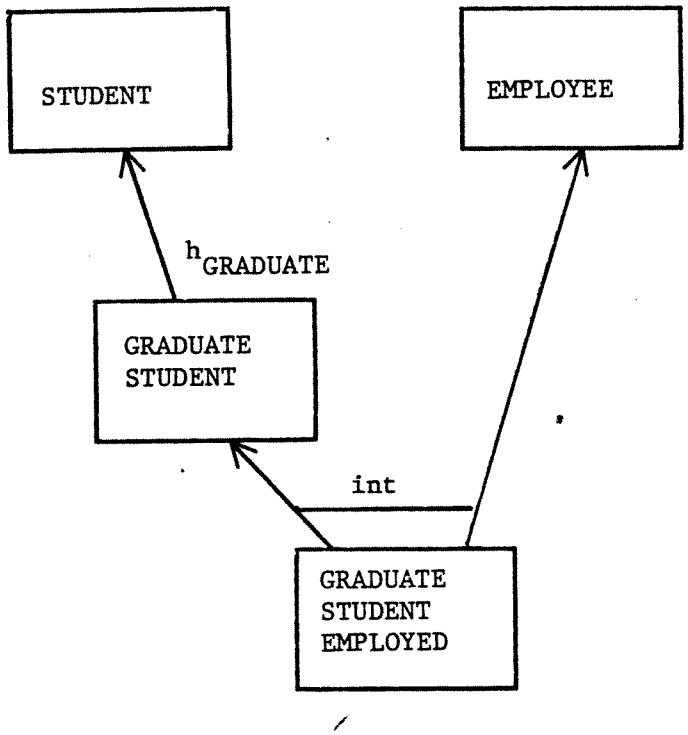


Figure 2. Example of a query graph

Figure 3a - Query graphs for q_1 and q_2

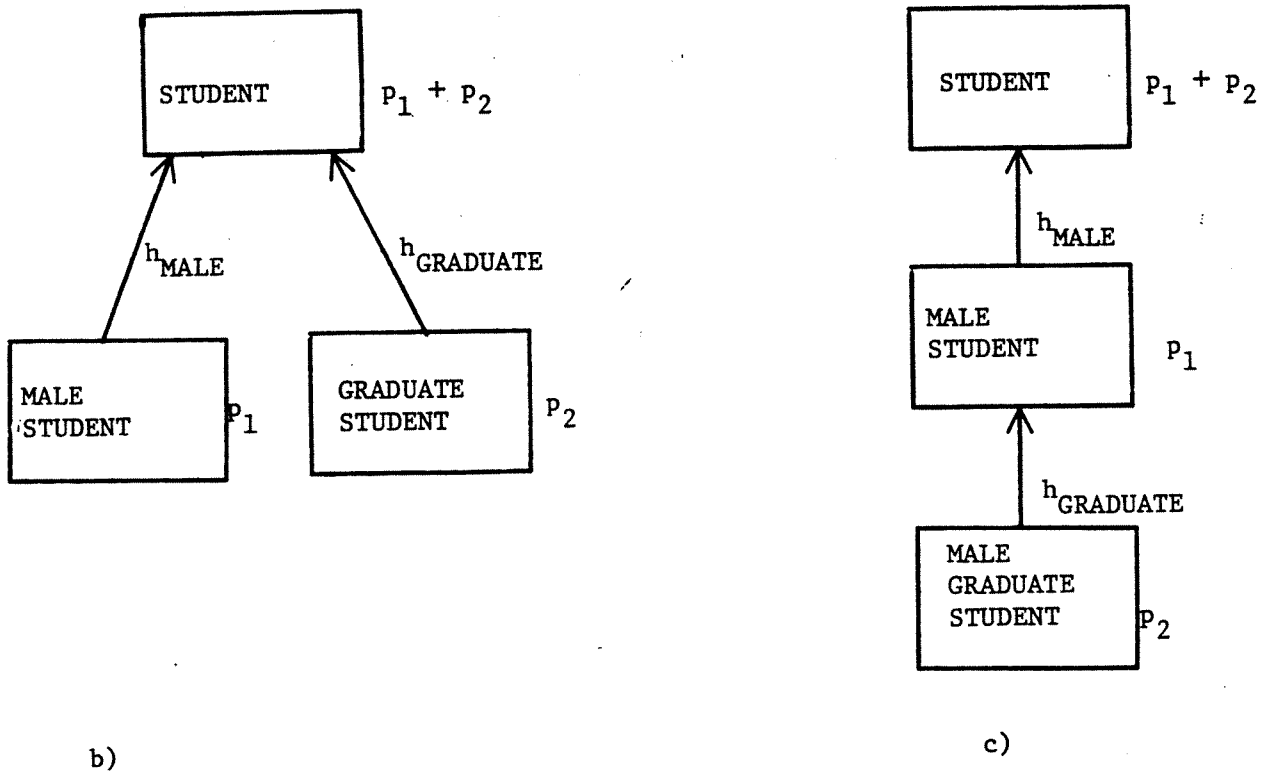


Figure 3 - Merged query graphs for Figure 3a

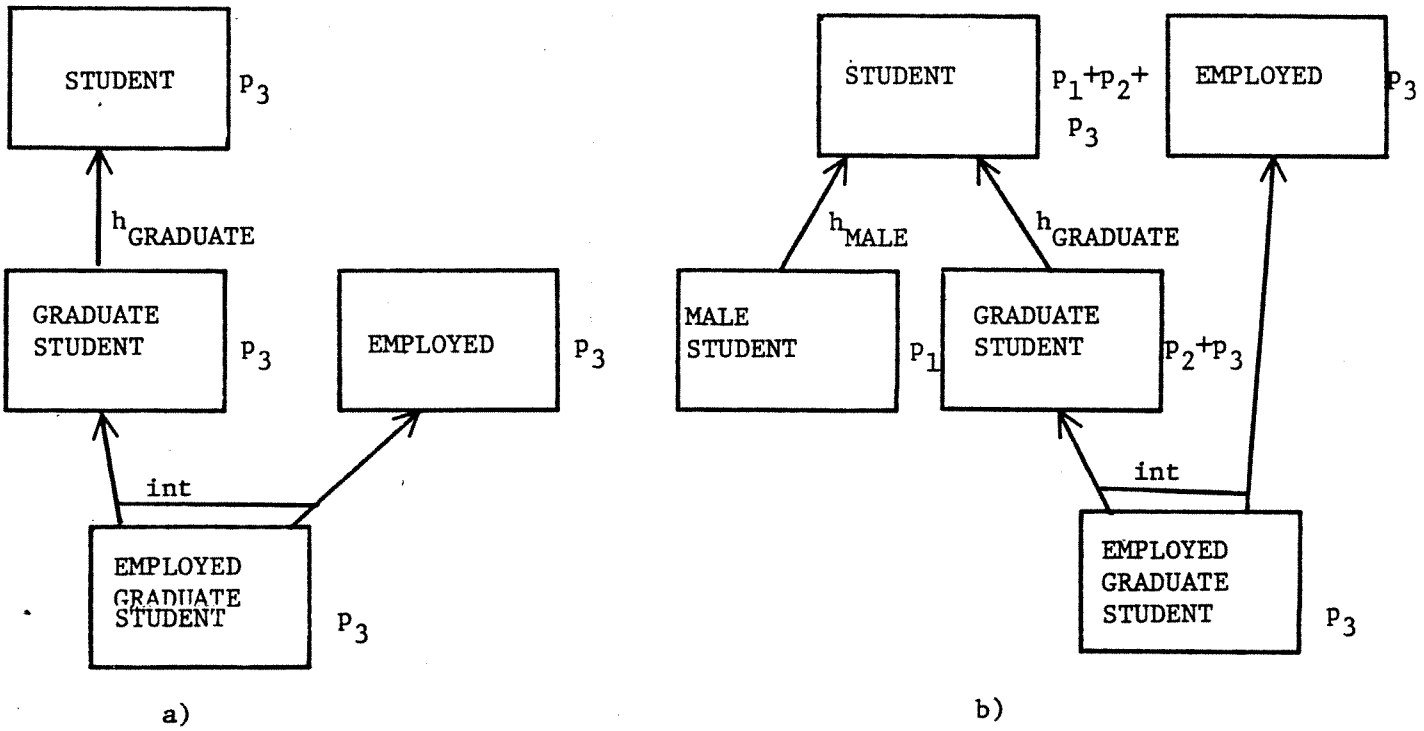


Figure 4 - Query graph for q_3 (a) and merged graph (b)

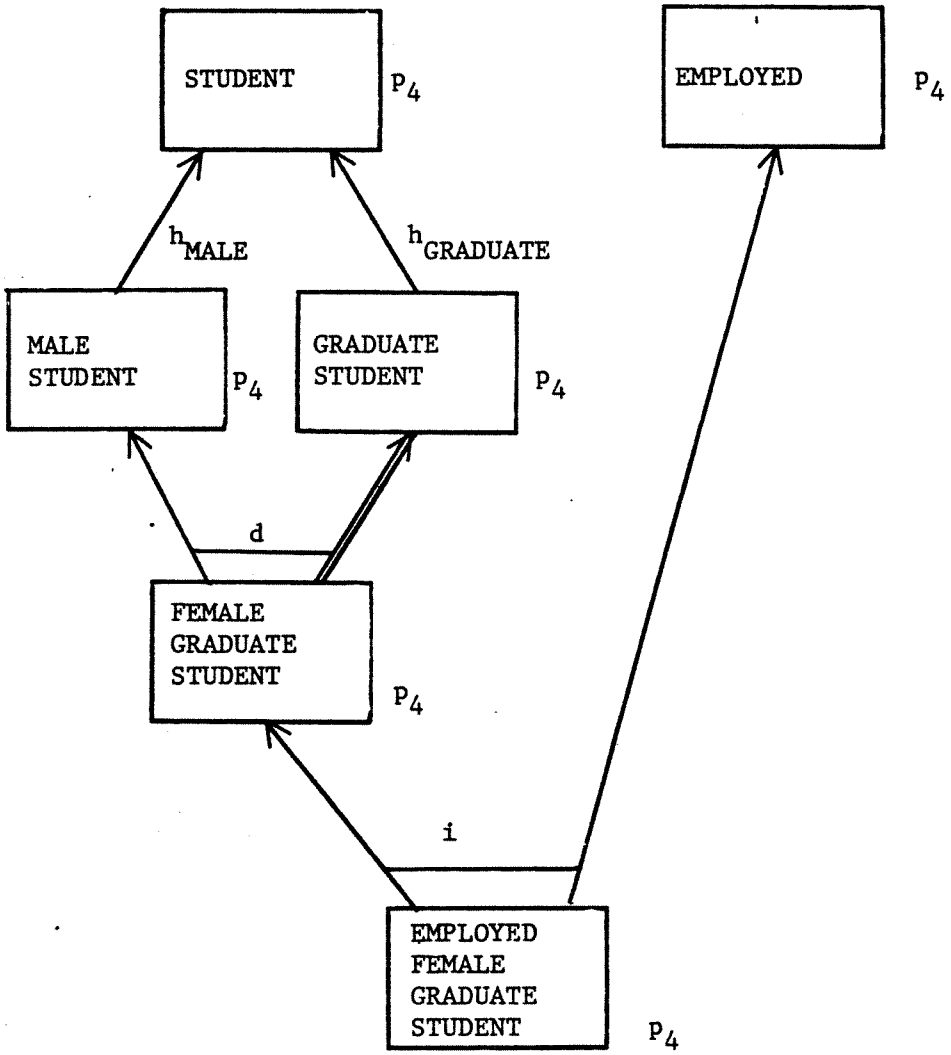


Figure 5a - Query graph for qúery q_5

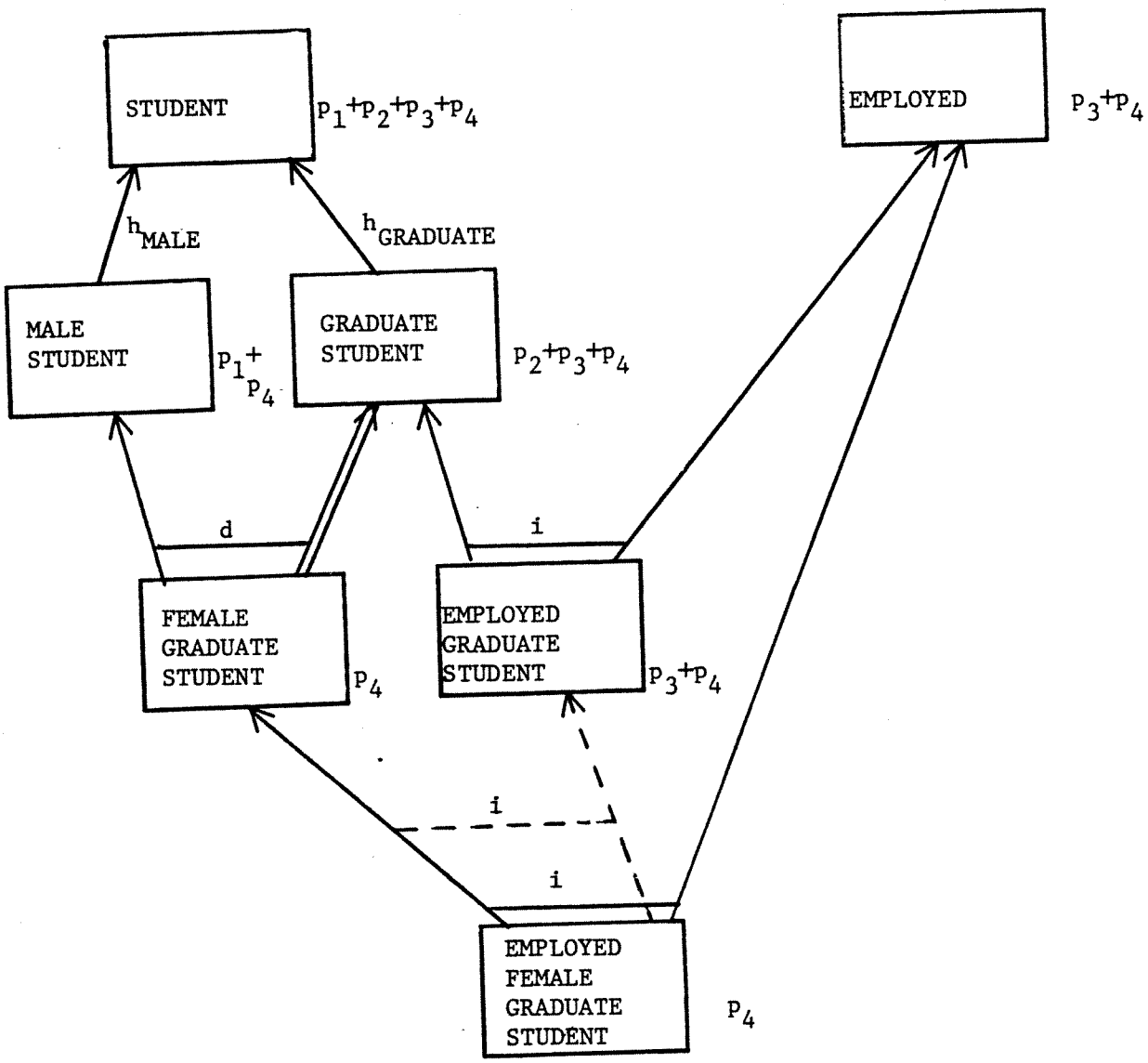
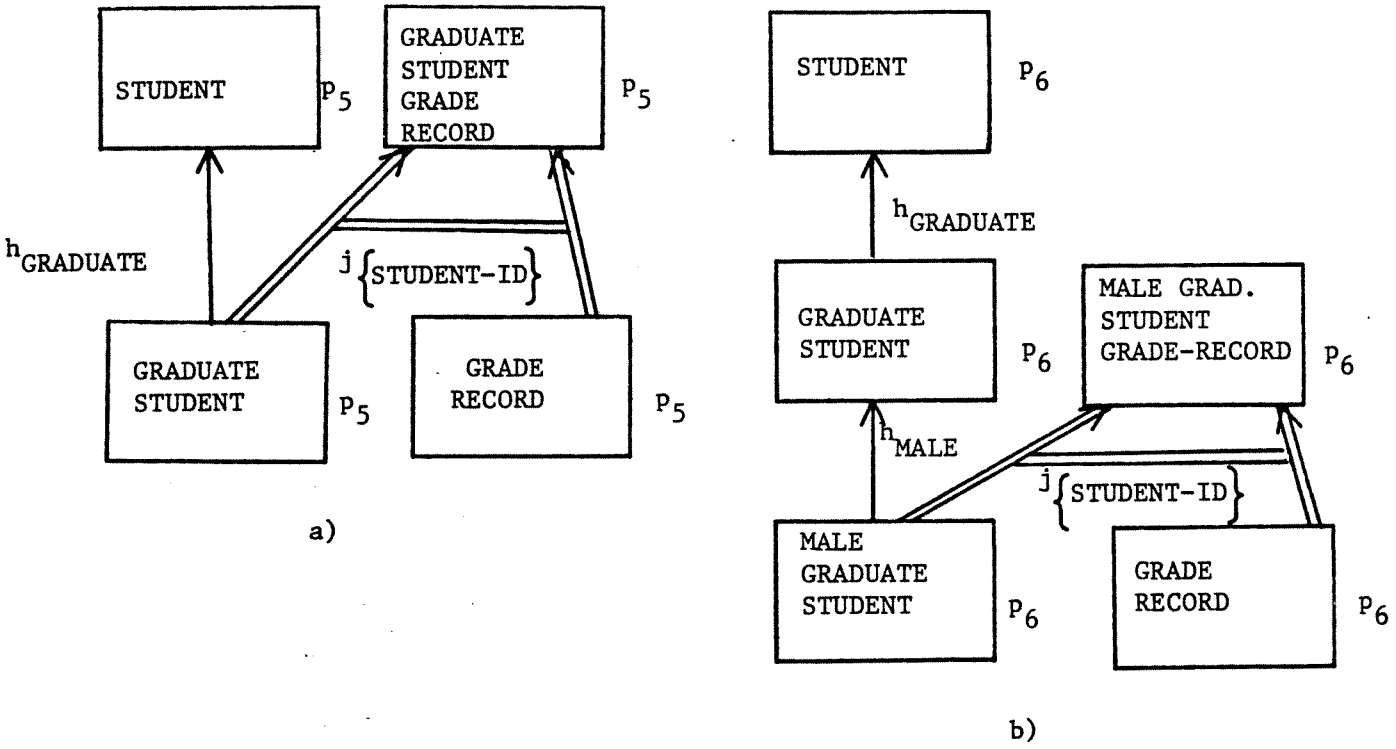


Figure 5b - Merged query grapg including the query in Figure 5a.



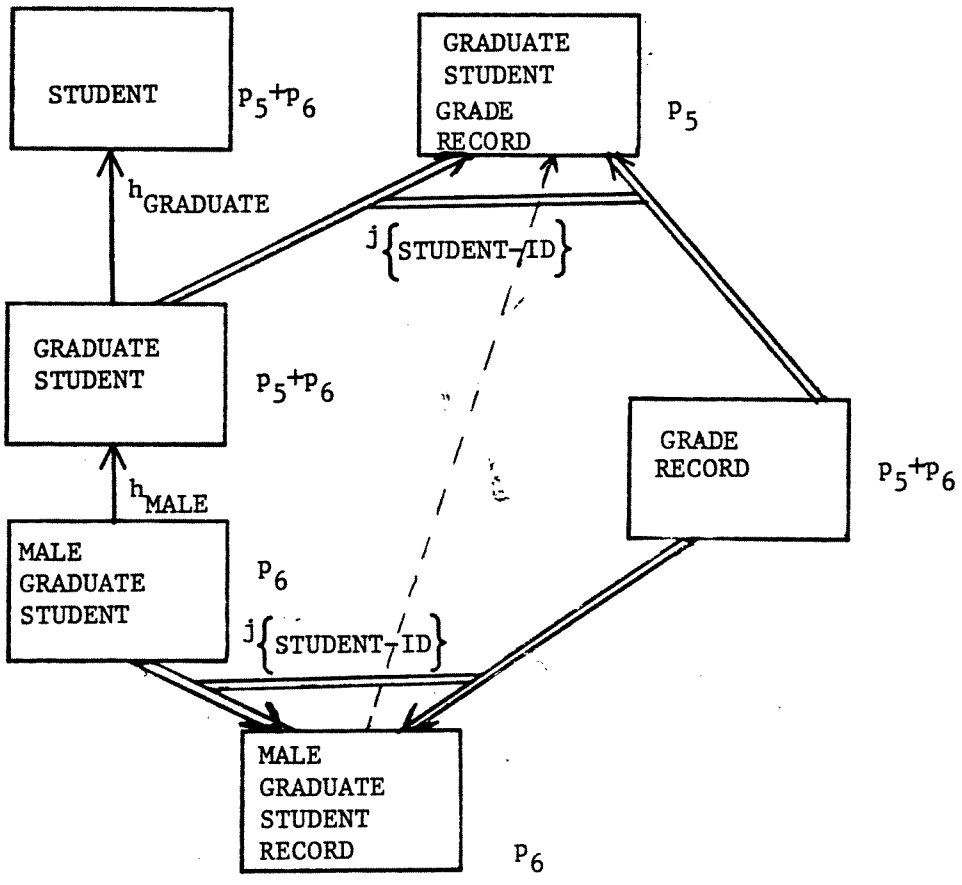
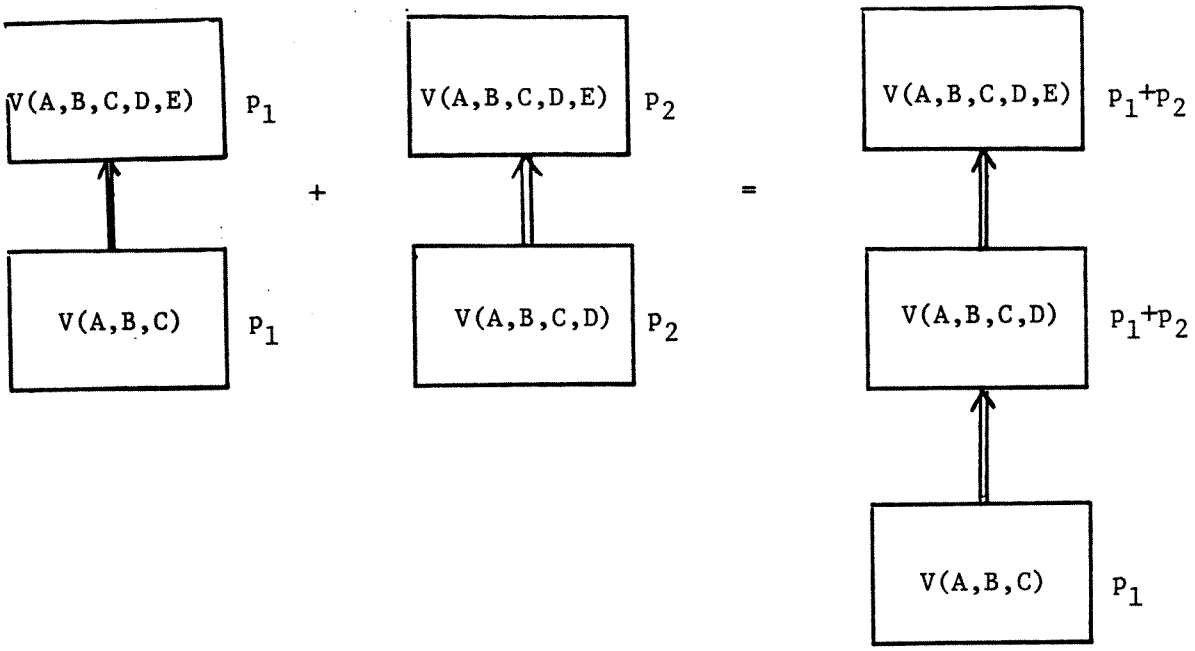
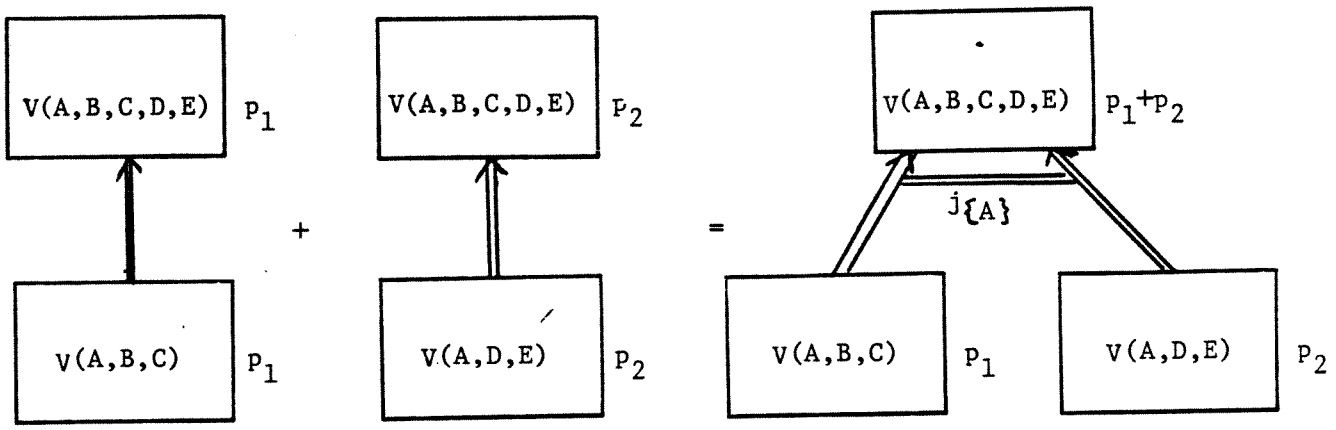


Figure 6 - Query graphs for q_5 and q_6 and merged graph



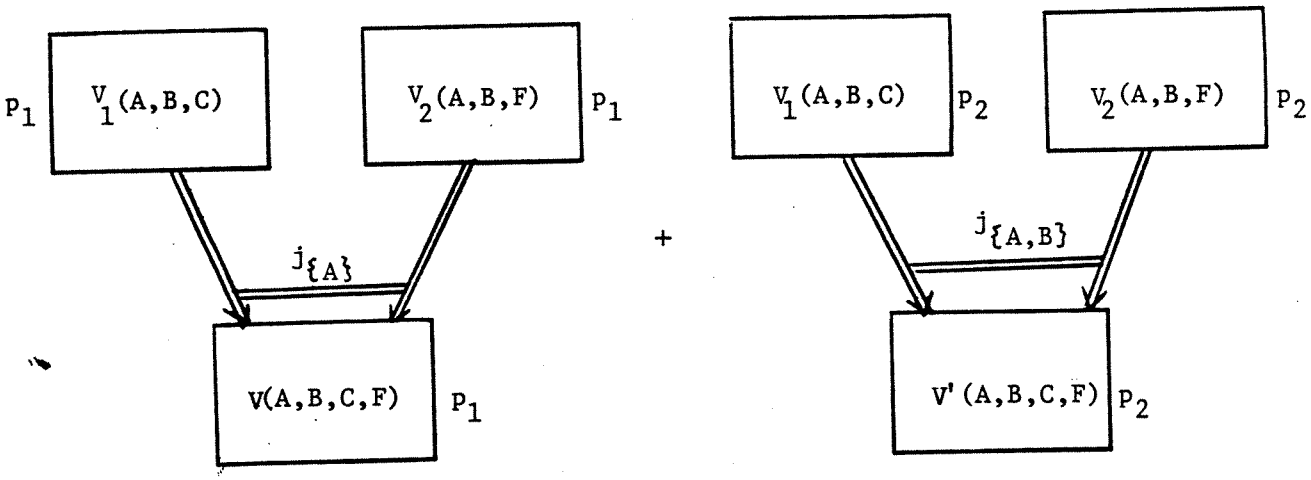
(a)

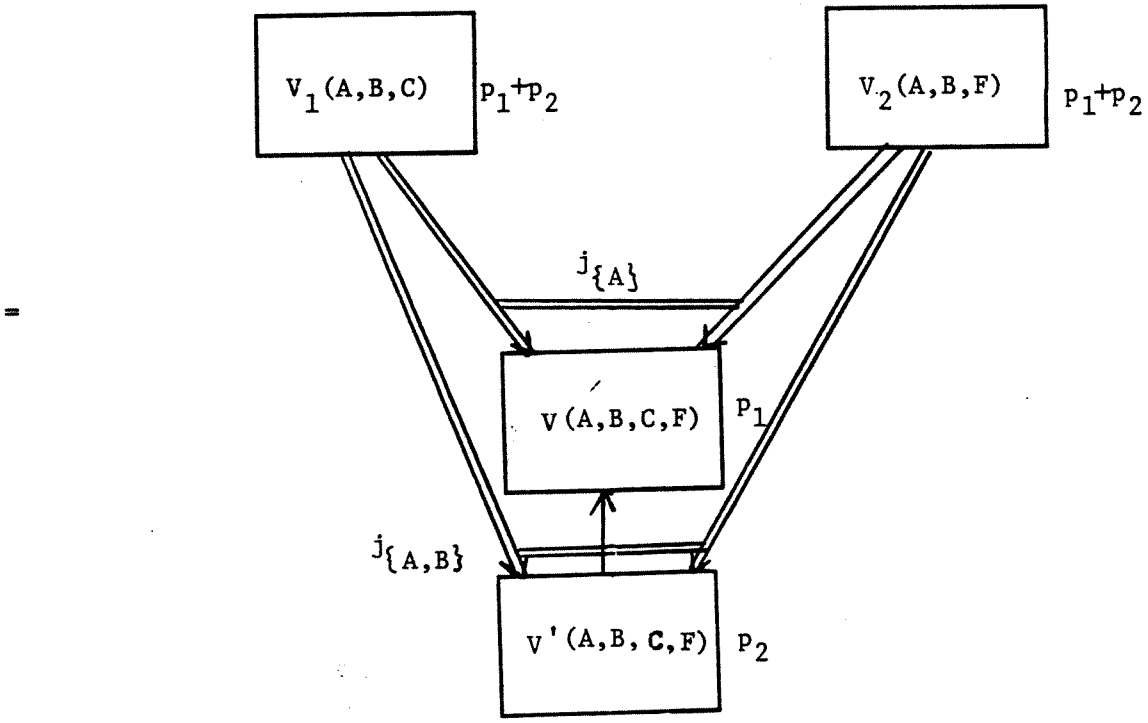


(b)

"A" must be a key of V for this merge to be legal

Figure 7





(c)

Figure 7 (cont.)

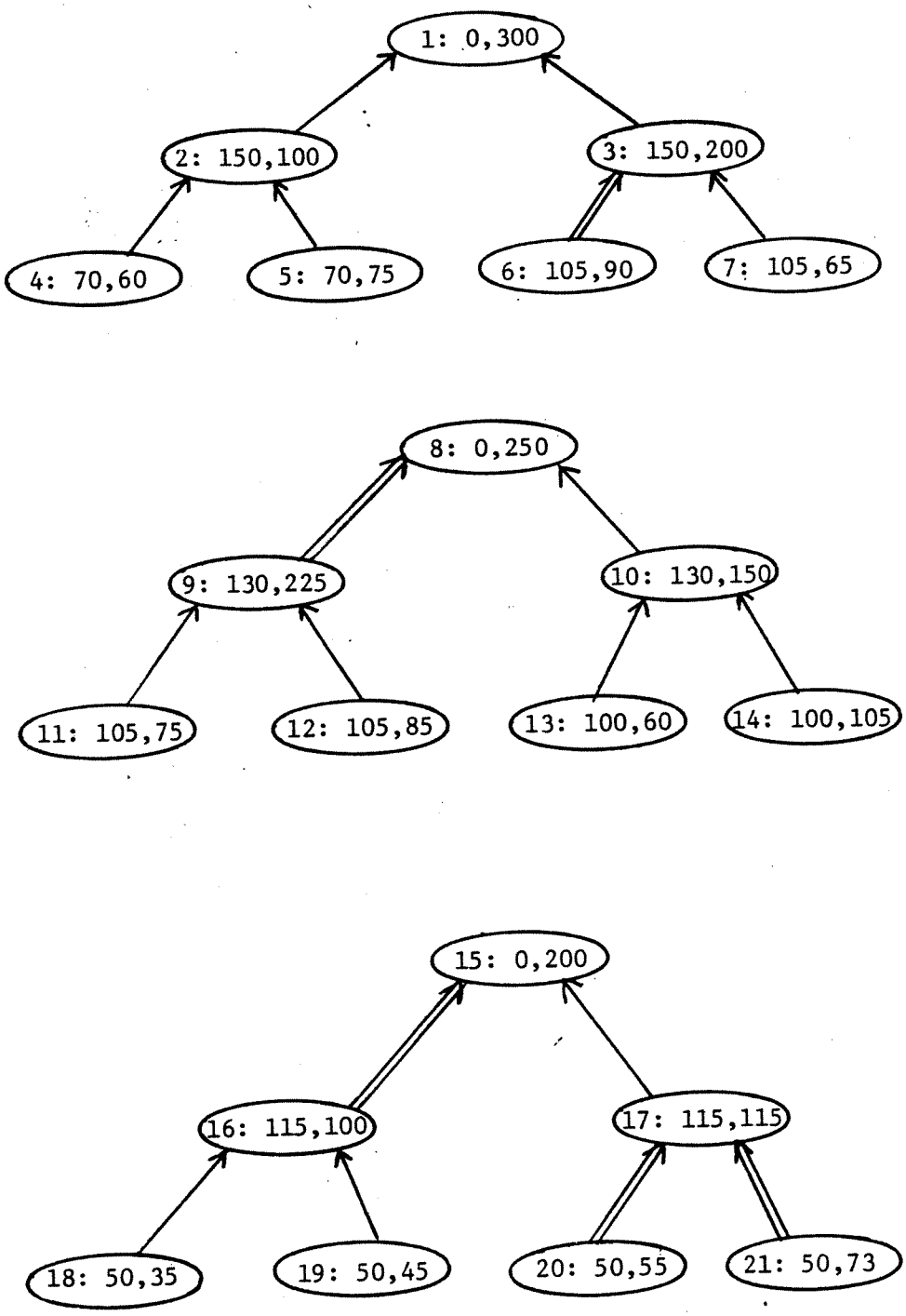

1: 0,300
2: 150,100
3: 150,200
4: 70,60
5: 70,75
6: 105,90
7: 105,65
8: 0,250
9: 130,225
10: 130,150
11: 105,75
12: 105,85
13: 100,60
14: 100,105
15: 0,200
16: 115,100
17: 115,115
18: 50,35
19: 50,45
20: 50,55
21: 50,73

Figure 9