A GRAPH REPERSENTATION OF A SEMI-JOIN PROGRAM AND ITS OPTIMIZATION

LYDIA LAI-CHU LUK


CENTRE FOR NFLD. STUDIES
JUN 27 1984
MEMORIAL UNIVERSITY
OF NEWFOUNDLAND

CANADIAN THESES ON MICROFICHE

I.S.B.N.

THESES CANADIENNES SUR MICROFICHE

National Library of Canada
Collections Development Branch

Canadian Theses on
Microfiche Service

Ottawa, Canada
K1A 0N4

Bibliothèque nationale du Canada
Direction du développement des collections

Service des thèses canadiennes
sur microfiche

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us a poor photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de mauvaise qualité.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE

NL-339 (r. 82/08)

Canada

A GRAPH REPRESENTATION OF A SEMI-JOIN PROGRAM AND ITS OPTIMIZATION

by

© Lydia Lai-Chu Luk, B.Sc.

A Thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science

Department of Computer Science
Memorial University of Newfoundland
September 1982

St. John's Newfoundland

ABSTRACT

A semi-join program is considered to be a strategy for processing a query in a distributed database system. To produce such a strategy for a general query, heuristics are required, as the problem of deriving an optimal solution is considered to be difficult. This thesis presents a semi-join program in graph-theoretic terms. In this framework, it seeks to improve the semi-join program by transforming it into one with non-increasing cost and non-decreasing benefit. The transformation algorithm runs in O(n**3) time where n is the number of semi-joins in the program.

ACKNOWLEDGEMENTS

I wish to thank my supervisor Dr. K. Vidyasankar for his invaluable guidance of my research. I wish to express my gratitude to Prof. Jane Foltz for her help throughout the duration of my M.Sc. program. Thanks are due to the internal examiner, Dr. Bill Day and the external examiner, Dr. Jean Tague of the University of Western Ontario, for their suggestions to improve this thesis. My husband, Wo-Shun, has been a constant source of moral and technical support, without which this thesis would never have become a reality.

Table of Contents

List of Figures

Figure — Page

Chapter 1

INTRODUCTION

In a computer network environment, a database may be distributed to multiple geographically different locations (sites). In comparison to the centralized database with remote accesses, one of the advantages of the distributed database is that data traffic, and hence cost of data communications, can be much reduced. In many application environments, some portions of the database tend to be accessed mostly from a few particular sites. Shipping portions of a database to where they are most often needed will reduce the cost of data communications. Nevertheless, no matter how well the database is distributed there are still queries that have to reference data stored in more than one site. We shall study the problem of constructing an efficient method to answer this type of query. This problem is particularly important when the network is implemented on the pay-as-you-use type of public data networks. The obvious but primitive method is to transmit all required data directly from various sites to the site where information is needed, and process it there. This has been shown to be far from optimal in terms of data communication cost [Hevner & Yao, 79]. Semi-join [Bernstein & Chiu, 81] is a pre-processing operation for a join operation that requires data from two different sites. It can prevent some superfluous

data from being sent to the result site, thus reducing the cost of data communications. Within this framework, the design of a strategy to process a query is a matter of choices of semi-joins and their execution sequence.

Several papers have addressed this problem ([Hevner & Yao, 79], [Chiu & Ho, 80] and [Yu et al., 80]). They construct query processing strategies which are optimal for some restricted classes of queries. For a slightly more general class of queries, [Hevner, 79] has shown the problem of deriving an optimal strategy is NP-complete. Thus a heuristic approach is necessary to deal with an arbitrary query.

Two algorithms have been proposed by [Bernstein et al., 81] and [Hevner, 79], based on different heuristics. Let us define a set of semi-joins to be performed in a certain order as a semi-join program. In [Hevner, 79], the algorithm derives a set of semi-join programs which are supposed to be executed independently. The communications cost calculation in each program does not take into account the reduction in data as a result of prior execution of another semi-join in a different program. By merging these programs together and re-arranging them, a substantial cost reduction can be realized. A similar idea is shared by [Bernstein et al., 81], who suggest that after a semi-join program has been

derived, an improvement can be obtained by reversing two semi-joins in their execution sequence. The objective of this thesis is to generalize this idea further and to present a generalized algorithm with polynomial time-bound to optimize a given semi-join program by converting it into another semi-join program with strictly lower communications cost. This approach will partly compensate for any lack of backtracking capability of the heuristic which produces the original semi-join program. The algorithm also distinguishes semi-joins that can be processed in parallel without increasing the total processing cost. It will be shown the transformed semi-join program is optimal in some sense.

The remainder of this thesis is organized as follows. Background information of this thesis is contained in Chapter 2, which describes precisely the model and the necessary terms we shall adopt for subsequent discussion. A graph representation of the semi-join program is introduced in Chapter 3 which sets up a framework for optimization. Chapter 4 discusses the rules that the transforming algorithm has to follow and their implications. Chapter 5 discusses binding and forking as optimization techniques. It also presents an algorithm which constructs a query processing strategy, together with a discussion of its validity and timing analysis. Chapter 6 discusses the optimality of the transformed semi-join program. Chapter 7

is the summary of the thesis.

Chapter 2

BACKGROUND

We assume the database is distributed without overlap to a collection of sites which are fully connected by a communication network; that is, data can be transferred directly between any two sites. The disjoint portion of the (global) database located in each site is itself a (local) database and managed by a local database management system (DBMS). Request of information about the global database in the form of a query can be initiated at every site. The query can either be processed by the local DBMS if it is equipped with a (global) data dictionary containing data characteristics of each local database or be shipped to a site where such a data dictionary is available. In either case, the processing of the query results in a sequence of operations which cause data operations to be performed by local DBMS's, or data to be transmitted from one site to another. At the end of execution of this sequence of operations, the necessary information is sent to the site where the information is needed. We shall call this site the result site.

For simplicity, we view logically the global and local databases as relational databases. A relational database consists of a number of relations, each of which is a two-

dimensional table. The columns of a relation are labelled by a set of distinct attributes. To distinguish one attribute of one relation from the same attribute in another relation, we identify an attribute first by the relation name followed by the attribute name. For example, R.A refers to the attribute A in the relation R. The set of all possible values as entries in a column is called a domain. A row of a relation is called a tuple. For a relation name R, there can be two or more relations. We call them occurrences of R. In this thesis, there can be only one occurrence of a relation name at any one time. The operations that are performed by a relational DBMS are: selection, projection and join. The exact definitions of these operations can be found in [Date, 81].

A query consists of a qualification and a target list. The target list specifies the attributes of interest to the user. Qualification is a conjunction of selection clauses of the form (R.A = constant) or (R.A = R.B) and join clauses of the form (R.A = S.B). The selection operation eliminates those tuples in a relation which do not satisfy any selection clauses. The join operation merges two relations specified in the join clause together to create another relation and in the process eliminates tuples whose entries in the attributes do not satisfy the join clause. The projection operation eliminates those attributes in the

relation that are not needed in any operation and do not appear in the target list.

An answer to a query is a relation containing all the tuples, and only those tuples, for which the qualification of the query is true.

We shall assume, as in most of the papers previously cited, that the cost of local processing is insignificant compared with the cost of data communications and is therefore ignored. This assumption also simplifies the model. As all local operations can be first performed to eliminate any superfluous data transfer, we shall assume from now on that only one relation resides in each site and the (global) database has as many relations as number of sites. The qualification of a query now consists of only join clauses involving relations in two different sites.

There are joins that do not correspond to any join clauses in the qualification, but nevertheless are just as "legal". We adopt the concept of legality of a join operation from [Bernstein et al., 81]. A join is legal for a query if execution of the operation will not alter the answer. To find all the legal operations, we use a join graph to represent the qualification of a query. For a given query, a join graph is an undirected graph JG = (V, E),

where a node in V represents one attribute of a relation in the qualification and an edge in E represents a join clause. For each connected component in the JG, we assign a distinct label. An attribute in relation R will be relabelled as R.A if it belongs to a connected component with label A. The transitive closure of JG, denoted by JG+, is a graph which has the same set of nodes as JG, with an edge for every pair of nodes which are connected by a path in JG.

Two queries are equivalent if their answers are the same irrespective of the forms in which they appear [Bernstein & Chiu, 81]. An important theorem about equivalence of queries is stated below without proof.

Theorem 1: Queries with the same target list, are equivalent iff their join graphs have the same transitive closure.

Proof: see [Bernstein & Goodman, 79] Q.E.D.

Thus a join is legal iff the corresponding join clause is represented in JG+.

In this fully connected communication network where data can be transferred between any pair of sites, distance between them is assumed to have no bearing on the transmission cost. In fact, the cost is dependent on only two factors:

(i) start-up cost, which covers the expenses incurred by the communication protocols, and

(ii) amount of data transferred, on the principle that the more data is transferred the higher is the cost.

Semi-join, as introduced in [Bernstein & Chiu, 81], is a relational operation that can result in cost reduction in performing a subsequent join operation. Consider a join operation corresponding to the join clause R.A = S.A, where R and S are different relations (and hence residing in different sites) and A is the join attribute. This operation results in elimination of all those tuples in R and S whose entries in the attributes in one relation are not equal to any entries in the same attribute of the other relation. In terms of data reduction, semi-join is "half" of a join. A semi-join of S by R over A is a join operation of R[A] (i.e. the projection of R over A) and S over join attribute A. It is denoted by an ordered pair (R, S). R and S are called respectively the left and right sides of the semi-join. There are two steps to perform this semi-join in a distributed environment:

(i) send entries in the attribute A of R to the site where S resides,

(ii) Treating the entries received as constituting a relation R', perform a join operation of R' and S.

The net result of this operation is the removal from S of all the tuples which do not satisfy the join clause, while R remains unchanged. The cost of this operation according to

our model is the cost of shipping the data in step (i). The benefit obtained as a result is the data reduction in S in step (ii). As in general the reduced S has to be sent to the result site for final assembly (i.e. the join operation with R and projection on the target list), the semi-join will be a worthwhile (or cost-beneficial) operation if the benefit exceeds the cost.

Since the semi-join is subsumed by the join, a semi-join is legal iff the corresponding join is. As we are only interested in legal semi-joins, we shall exclude illegal semi-joins from our discussion and from now on, semi-joins refer only to legal semi-joins, unless otherwise specified.

For queries of a certain type, it is possible to derive the answer by using only semi-joins [Goodman & Bernstein, 79]. Optimization procedures have been devised for a subset of these queries which require only semi-joins [Chiu & Ho, 80], [Yu et al., 80]. We believe however that the class of the queries to which the procedures are applicable is too restrictive in a general system. We assume that certain semi-join operations are performed as data reduction tactics and all the relations will be shipped to a site for final assembly of the result. The same approach is taken by [Hevner, 79], [Bernstein et al., 81] and [Yu et al., 82]. The query processing algorithm OPT as described in

[Bernstein et al., 81] has a semi-join program as the output, while the algorithm GENERAL as described in [Henver, 79] derives sequences of semi-joins, one for each relation (site). These sequences are supposed to be executed in parallel. To facilitate cost calculation, it is convenient to merge these semi-join programs into one single program, taking into consideration the timing of overlapping semi-joins.

Both algorithms OPT and GENERAL employ heuristics, and their outputs are suboptimal with respect to total cost. This thesis seeks to design an algorithm which transforms the semi-join program into a "better" one in the sense it has lower total cost and greater total benefit. (Note that the total benefit, i.e. the total data reduction resulting from the execution of the semi-join program, is actually the cost saving in the subsequent shipping of all the relations to the result site.) If no assumption is made about the start-up cost in comparison to the cost of actually transmitting the "useful" data, the reconstructed semi-join program has to have the same number of semi-joins. The technique used here is to re-arrange the semi-joins in the original program and possibly replace some of them by others with lower cost, or greater benefit or both. The details of this method will be described in Chapters 3 & 4.

Chapter 3

GRAPH REPRESENTATION OF A SEMI-JOIN PROGRAM

In this chapter we shall develop a graph representation of a semi-join program which will be used as a framework for discussing optimization of the semi-join program. But first, let us formally define semi-join and the related concepts.

Semi-join of S by R over join attribute A is a join operation of R[A] and S over the join attribute A. S is then said to be reduced by R over join attribute A. Let the resulting relation be denoted by S'. In the context of this thesis, R and S reside in different sites, and S' is found in the same site as S. The cost of this semi-join is the cardinality of R[A] and the benefit is the difference in cardinalities between S and S' (obviously, $S' \subseteq S$). A semi-join program is a number of semi-joins arranged in their execution sequence. The cost of a semi-join program is the sum of the costs of all the semi-joins in it, while the benefit is the difference between the sums of cardinalities of all relations before and after the execution of the program.

As the semi-join program P is actually a query processing strategy, which specifies not only which semi-joins are to be executed, but also their execution sequence,

we shall call the graph representation of P a strategy graph. We shall explain in detail what we mean by a stratgey graph. An example strategy graph can be found in Appendix A.

In general, we could use nodes to represent relations and edges to represent semi-joins between two relations. However, the cost and the benefit of a semi-join depend on the occurrences of the relations at the moment the semi-join is executed. Moreover, to represent an executable program, the strategy graph must not contain any cycles. This is made possible by allowing occurrences of a relation to be represented by different nodes. Each node in the graph then represents an occurrence of a relation. Each node is denoted by a label and a relation name of the relation it represents. For example, N1(R1) and N2(R1) represent two possibly different, occurrences of R1. In this way, it is clear from notation as to which semi-join an edge represents and its cost and benefit. An edge represents a semi-join between occurrences of two different relations represented by the end nodes of the edge, and is given a label which is the join attribute of the corresponding semi-join. The edge is directed from its predecessor node to its successor node, in the same sense as the semi-join it represents. That is, after the semi-join is executed, (or equivalently, the edge is traversed), the occurrence in the preceeding node of the edge does not change, while the succeeding node represents

the resulting occurrence of the relation on the right side of the semi-join. More precisely, an occurrence of a relation is determined recursively as follows. The first occurrences of all relations without any predecessor nodes (i.e., no in-coming edges) are the same as the original relations before the semi-join program is executed. If a node N1(R1) has at least one predecessor, then N1 is the same as the last occurrence of R1 immediately before N1, minus the tuples that may have been eliminated as a result of executions of all the semi-joins which are represented by the in-coming edges at N1. (If N1 is the first occurrence of R1, then the 'last occurrence of R1 immediately before N1' is the original R1).

There are two types of relationships between a pair of nodes: predecessor-successor relationships and parallel relationships. If there is a path from one node to another, then the former is a predecessor of the latter. Otherwise they are parallel to each other. A node is a last successor of a certain node if it has no successors. An edge is a successor of another edge if it can only be traversed after the other edge has been. Dummy edges are allowed in the strategy graph. They play a role similar to dummy activities in the critical path method (CPM). A dummy edge does not represent a semi-join and its only purpose is to establish a predecessor-successor relationship between a pair of nodes

or edges.

An edge is traversed when the corresponding semi-join is executed. The cost (or benefit) of an edge is that of executing the semi-join involving the occurrences represented by the end nodes. Dummy edges of course cost nothing and provide no benefit. The traversal of the entire graph (i.e. execution of the corresponding semi-join program) amounts to traversals of all edges in the graph exactly once, such that the predecessor-successor relationships of the edges are observed. Thus the cost of the entire strategy graph is the sum of the costs of all the edges. The benefit of the graph is the amount of reduction in all relations. Although there is more than one way to traverse a strategy graph in terms of the serial order in which the edges are traversed, there is only one cost and one benefit, as will be shown in Lemma 3.

The strategy graph as described above is a suitable representation for a semi-join program. It is true that a semi-join program by definition is a linear ordering of semi-joins. Nevertheless, two 'unrelated' semi-joins, eg. semi-joins referring to disjoint pairs of relations, may be executed in either order with identical cost and benefit. Thus as far as cost/benefit is concerned, a semi-join program is actually a partial ordering of semi-joins, hence

an ideal candidate for graph representation. In other words, the semi-joins in a semi-join program may be executed in different linear order from the one implied in the program and yet the execution yields identical cost/benefit. The ordering of the semi-joins is subject to the following conditions: (a) there must not be cycles, and (b) the occurrences of the same relations must be linearly ordered. In addition, 'related' semi-joins should be linearly ordered, if the graph is meant to represent 'faithfully' the semi-join program, i.e. to have the same cost/benefit. Algorithm 1 below shows how to construct such a graph. It is significant not only in demonstrating the validity of a strategy graph, but also in providing insight into its optimization, which is the job of Algorithm 2 in Chapter 5.

Based on a semi-join program P, with n semi-joins, the strategy graph can be constructed by the following algorithm with the underlying rationale stated in lemmas 1-3.

Algorithm 1:

DO i = 1 to n;
Let the ith semi-join in P be (R1, R2)

1. IF R1 and R2 are new (i.e. not represented by any existing nodes), THEN DO:
   Add a separate connected component with one edge to the graph as follows: create new nodes N1 and N2 for R1 and R2 respectively and add the edge (N1(R1), N2(R2)).

2. IF R1 is new but R2 is not, THEN for N2 being the last occurrence of R2, DO:
   2.1. IF N2 has at least one successor, THEN DO:
      (i) create new nodes N1 and N3 for R1 and R2

respectively;
(ii) link all last successors of N2 with dummy edges to N1; and
(iii) add an edge (N1(R1), N3(R2)) to the graph
2.2. IF N2 has no successors, THEN DO:
(i) create a new node N1 for R1; and
(ii) add an edge (N1(R1), N2(R2))

3. IF R2 is new but R1 is not, THEN for N1 being the last occurrence of R1, DO:
(i) create a new node N2 for R2; and
(ii) add an edge (N1(R1), N2(R2))

4. IF neither R1 nor R2 is new, THEN for N1 and N2 being the last occurrences of R1 and R2 respectively, DO:
4.1. IF N2 has at least one successor, THEN DO:
(i) create new nodes N3 and N4 for R1 and R2 respectively;
(ii) link all last successors of N2 with dummy edges to N3; and
(iii) add an edge (N3(R1), N4(R2))
4.2. IF N2(R2) has no successors, THEN DO:
(i) add an edge (N1(R1), N2(R2)).

The algorithm implies that the "last" occurrence of a relation can always be determined unambiguously. This is in fact quite obvious from the algorithm itself. Every time a new occurrence of a relation which has already been represented, is added to the graph, this occurrence will be made a successor of all existing successors of the previous occurrence of the relation. Thus, by a simple inductive argument, we can establish the following lemma:

Lemma 1 Any two nodes representing the same relation must bear a predecessor-successor relationship.

The fact that the traversal of a strategy graph is feasible is shown below:

Lemma 2 The strategy graph as constructed according to Algorithm 1 contains no cycles.

Proof: A cycle might be created only if there is a possibility that a new edge (dummy or otherwise) is incident on N, one of the existing nodes. This happens under conditions 2.2. and 4.2. In both cases, there are no successors of N, which means that there can be no path from N to any other existing node. Q.E.D.

There is a one-one correspondence between the semi-joins in the program and (non-dummy) edges in its strategy graph. The relations involved in a semi-join are the same as the relations represented by the end nodes of its corresponding edge. The only difference is in the execution sequence. While the semi-joins in the program must be executed in strict sequence as they appear in the program, parallel edges in the strategy graph can be traversed in an arbitrary manner. This leads to the question whether the semi-join program and its strategy graph are the same in terms of the cost and the benefit, the answer of which is given in the following lemma:

Lemma 3 The cost and benefit of a strategy graph as constructed according to Algorithm 1 are independent of the manner in which it is traversed (as long as the predecessor-successor relationship is observed), and are identical to the cost and benefit of the semi-join program on which the

strategy graph is based.

Proof: We prove the lemma by induction. Initially, there is a null graph. Assume as induction hypothesis that the assertion is true for the first i-1 semi-joins in P; that is, the cost and benefit of the graph constructed so far are the same as those of the part corresponding of P. In particular, the last occurrences of all relations in the graph are the same as the corresponding relations in P after executions of these semi-joins.

Consider the insertion of a new edge corresponding to the ith semi-join (R1, R2). The cost and benefit of executing this semi-join are determined by the occurrences of R1 and R2 at that moment. Moreover the execution of this semi-join will not affect any relations but R2. In order that the assertion be true for the first i semi-joins in P, we have to show that the predecessor and successor nodes of the new edge contain the last occurrences of R1 and R2. We also have to ensure that all existing edges with the last occurrence of R2 as a predecessor precede the new edge, because otherwise, with the new (possibly reduced) occurrence of R2, the costs and benefits of these edges may be different from the corresponding semi-joins in P. Let us now examine the conditions in Algorithm 1, under which the new edge will be inserted. The assertion is obviously true for condition 1, as both N1 and N2 are first occurrences of

R1 and R2 respectively. In condition 2, R1 is new and N1 will be the first occurrence of R1 with no in-coming, non-dummy edges. Thus it will be the same as the original R1. If the last occurrence of R2 has no successors, then it can be used as the successor node of the new edge. This is precisely condition 2.2. Otherwise, all the existing edges will have to be made the predecessor edges of the new edge, which is what condition 2.1 does. In condition 3, we use the last occurrence of R1 as the predecessor of the new edge and the new node for R2 as the successor node. According to the way an occurrence of a relation is determined, this new node contains the original R2, minus the tuples that may be eliminated as the result of execution of the semi-join. So the assertion is also true for condition 3. We use a combination of the arguments for conditions 2.1 and 3 to show the assertion is true for condition 4.1. Similarly we use a combination of the arguments for conditions 2.2 and 3 to show the assertion is true for condition 4.2. Thus we conclude that the assertion is true for all four conditions in Algorithm 1. Q.E.D.

Chapter 4

OPTIMIZING A SEMI-JOIN PROGRAM

Having set up the strategy graph as a representation of the semi-join program, we now seek improvement on the semi-join program in this framework. By Algorithm 1, a strategy graph can be constructed to be an equivalent of the program. But the very process of doing so reveals some easy ways to improve the semi-join program. For example, the dummy edge is an artificial device to make sure two edges are traversed in the same sequence as the corresponding semi-joins appearing in the program. A dummy edge is needed to present the semi-joins (R1, R2) and (R3, R1) in execution sequence. If we reverse the order of execution of these semi-joins in the program, no dummy edge is needed and R1 will be first reduced before R2 is reduced by the already reduced R1. Clearly (R1, R2) costs less when executed this way.

Specifically, let the original (input) semi-join program be P. We wish to design an algorithm which transforms P into another semi-join program P', such that the following conditions are satisfied:

c.1 P' and P give the same answer to the given query,

c.2 P' has a cost no greater than P <u>for all instances</u> of the relations, and

c.3 P' has a benefit no less than P <u>for all instances</u> of

the relations.

Besides, the transformation is subject to the following restriction:

r.1 Other than P itself, no additional information, such as the original query and the specific cost function, is available to the transforming algorithm.

We shall call transformations of this type lossless transformations. In terms of cost and benefit, a lossless transformation will produce a semi-join program which is no worse, and potentially better, than the original semi-join program.

According to Theorem 1, c.1 will be satisifed if P' contains only legal semi-joins implied by P, which are actually represented by the edges in the transitive closure of the join graph of P. Thus c.1 is equivalent to the following:

c.4 All semi-joins used in a lossless transformation must be legal.

Let us now examine the implications of criteria c.2-c.3 in light of the restriction r.1.

As there is no information about the start-up cost compared with the unit cost for data sent, c.2 is equivalent

to the following:

c.5 the number of semi-joins in P' has to be no greater than that in P, and the total amount of data sent by P' is less than or equal to that sent by P.

Since we do not know the sizes of the attributes in a relation that may be required to participate in the ultimate join operation, it is difficult to estimate the exact amount of data reduction (hence the benefit) on a relation due to a semi-join. Thus we will require that

c.6 the data reduction on each individual relation by P' is no less than by P.

Obviously, this requirement will satisfy c.3.

We emphasize the phrase "for all instances" in c.2 and c.3. Thus a statement is considered true if and only if it holds for all instances of the relations involved. This provides the basis for proving Lemmas 4 to 7 in the next chapter.

Chapter 5

BINDING AND FORKING

Binding and forking are the techniques we employ to optimize a semi-join program P. The two are very similar and they will be treated in almost the same way.

The process of adding an edge to the existing strategy graph with one of the existing nodes as a successor node of the edge, is called binding . A semi-join is said to be bound to a node if the semi-join is going to be represented by an edge with that node as the successor node. The meaning of the word is derived from the fact that when the predecessor node is a new node or an existing node in a different component, then binding actually binds two separate components together, although the same term is used even when both end nodes are in the same component.

Suppose there are two consecutive semi-joins in P, (R1, R2) and (R3, R1), both over join attribute A. By Algorithm 1, the two will be represented by two edges, (N1(R1), N2(R2)) and (N3(R3), N4(R1)), with the former as a predecessor of the latter connected by a dummy edge (N2(R2), N3(R3)). However, if we bind the second semi-join at N1, we will have edges (N3(R3), N1(R1)) and (N1(R1), N2(R2)), and in effect reverse the order of execution of the semi-joins.

Thus R1 will be first reduced before R2 is reduced by the already reduced R1, while the cost of traversing (N3(R3), N1(R1)) remains the same as traversing (N3(R3), N4(R1)). Clearly (R1, R2) costs less when executed in this order.

Binding can take another form which causes a semi-join to be replaced by another different semi-join and produces a better program. For example, let there be 4 edges: N6(R6) -> N5(R5) -> N4(R4) -> N3(R3) -> N2(R2), which represent semi-joins with labels A, B, B and A respectively (see fig. 1). Suppose the input semi-join under consideration is (R1, R2), with join attribute A. Clearly, we can bind this semi-join at N2. However, it will be more cost-beneficial to bind the semi-join on N3 instead of N2, even though it may mean the semi-join will now be (R1, R3). We show that this new semi-join is a legal one as follows. By virtue of (R3, R2) and (R1, R2), both with join attribute A, all three attributes R3.A, R2.A and R1.A are included in the join graph component with label A, of the join graph. (R1, R3) is represented by an edge of the transitive closure of this component, hence it is a legal semi-join. Though both N2 and N3 are valid nodes for binding, N3 is better than N2. In principle, the earlier the binding, the better the resultant graph. Note that binding at N6 (i.e. (R1, R2) to be replaced by (R1, R6)) is illegal. The data reduction on R6 in N6 by R1 may not be passed onto R2 in N2, as there is an in-between node

N4 with B as the only join attribute.

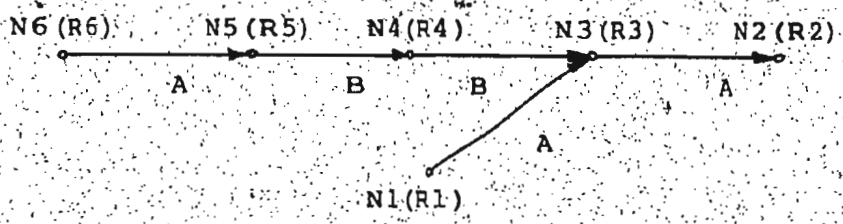


Fig. 1: An instance of early binding

We can now characterize binding in a more general way. A node N is A-connected to another node V if and only if there is a directed path from N to V, consisting of edges whose labels are A. A_PRE(N(R)) is a set of nodes which can be defined recursively as follows: (1) N is in A_PRE(N) and (2) if there is an occurrence of a relation say V(R) in A_PRE(N), all previous occurrences of R prior to V and all the predecessors of V which are A-connected with V(R) are in A_PRE(N). Note that the nodes in A_PRE(N) are not necessarily A-connected. Consider the following strategy graph:

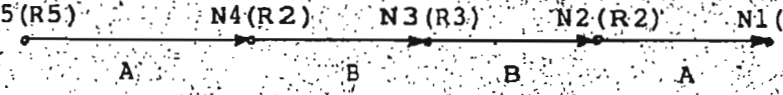


Fig. 2: An example of A_PRE(N1(R1))

Here, A_PRE(N1) consists of N1, N2, N4 and N5. Note that not all these nodes are A-connected to each other. Unlike in Fig. 1, the data reduction on R2 in N4 can be passed onto R2 in N2 and then onto R1 in N1.

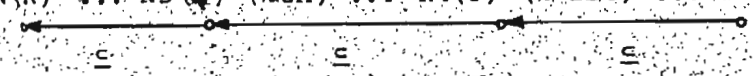
If a semi-join with join attribute A is to be bound at N, then A_PRE(N) represents all the nodes at which the semi-join can be bound so that the cost will not increase but the benefit for each relation involved will increase. For example, if V(R) in A_PRE(N) is chosen as a node for binding, then the benefit of the semi-join that R will receive will be passed on to every occurrence on the path from V to and including N. On the other hand, if the semi-join is bound at a node not found in A_PRE(N), then it is not guaranteed that the benefit for the relation in N (i.e. data reduction) will increase for all instances of the relations involved. This idea is formalized in the following two lemmas:

Lemma 4: Let N1(S) be an occurrence of some relation S in the strategy graph. Then N(R)[A] $\subseteq$ N1(S)[A] if and only if N1 is in A_PRE(N).

Proof:

IF part We prove this part by induction. The lemma is obviously true for nodes A-connected to N. Let us define, for the sake of this proof, an ordering of the occurrences of a given relation T in A_PRE(N). These occurrences are ordered such that N3, the kth occurrence of T, will be a successor of the (k+1)st occurrence, N4. Then N3(T)[A] $\subseteq$ N4(T)[A]. Now assume the lemma is true for all the nodes A-connected to and predecessors of any ith occurrence of T, $1 \leq i \leq k$. This means N(R)[A] $\subseteq$ N3(T)[A]. We shall prove that

this lemma is also true for any node, say N2(S), A-connected to and a predecessor of N4(T). This means N4(T)[A] $\subseteq$ N2(S)[A]. Hence, N(R)[A] $\subseteq$ N3(T)[A] $\subseteq$ N4(T)[A] $\subseteq$ N2(S)[A]. So this part of the lemma is true.



ONLY IF part If N(R)[A] $\subseteq$ N1(S)[A] and there exists no N2(T) for any occurrence N2 of any relation T such that N(R)[A] $\subseteq$ N2(T)[A] $\subseteq$ N1(S)[A], then either S=R, which means N1 is just another occurrence of R, or there exists a semi-join (S, R) with join attribute A. In either case, N1 is in A_PRE(N).

Thus let us assume there exists at least one occurrence N3 of some relation V such that the N(R)[A] $\subseteq$ N3(V)[A] $\subseteq$ N1(S)[A]. Without loss of generality, let N3(V)[A] be the largest column of A that satisfies the above assumption. Thus there will be no N2(T) for any occurrence of N2 of any relation T such that N3(V)[A] $\subseteq$ N2(T)[A] $\subseteq$ N1(S)[A]. Then the argument in the above paragraph applies to V and S (instead of R and S above). Thus N1 is in A_PRE(N3). By induction, N3 is in A_PRE(N) so that N1 is in A_PRE(N).

Q.E.D.

Lemma 5: Suppose a semi-join (R1, R2) with join attribute A is to be bound at node N3(R3). Then binding the semi-join at node N4(R4) instead guarantees non-decreasing benefit if and

only if N4(R4) is in A_PRE(N3).

Proof: The part of the Lemma regarding legality of the binding follows directly from Theorem 1 and the definition of A_PRE(N3).

It is easy to see that binding the semi-join at N3(R3) results in the same occurrence of R3 in N3 as binding at N4(R4) does, if N4(R4) is in A_PRE(N3). Extra benefit is derived from reduction in R4 and other relations in the nodes between N3 and N4.

On the other hand, if the semi-join is bound at a node N5(R5) which is not in A_PRE(N3), then by Lemma 4, it is not always true that N3(R3)[A] $\subseteq$ N5(R5)[A]. Consequently, binding at N5(R5) may not always provide the same reduction in R3[A] as binding directly at N3 will. Q.E.D.

A bind node of A_PRE(N) is a node in A_PRE(N) such that no other nodes in A_PRE(N) are predecessors of this node. Note that this definition can be extended to cover subsets of A_PRE(N). This is sometimes necessary because some bind nodes might create cycles in the graph and as such, have to be excluded from consideration. There may be more than one bind node for N. If V is a bind node, then A_PRE(V) consists of only V itself. Thus binding at a bind node will provide "optimal" benefit, in the sense that there is no other node

at . which binding is always more beneficial, according to Lemma 5.

Binding of a semi-join can lead to re-binding of the semi-joins processed before it. Re-binding an existing edge is to relocate the position of the successor node of the edge so that its new position is a bind node of the old position. A situation where re-binding occurs is described as follows. Let there be two edges (N3(R3), N1(R1)) and (N3(R3), N2(R2)) with labels B and A respectively (fig. 3a). Let (R1, R2) with join attribute A be the semi-join to be added to the graph. The new edge representing it has to be (N1(R1), N2(R2)) because binding at N3 would create a cycle (fig. 3b). After insertion of this edge, the edge (N3(R3), N2(R2)) is no longer "optimally" bound, since A_PRE(N2) contains N1 as well. By the principle of early binding, N1(R1) will be a bind node of A_PRE(N2), assuming no other predecessor edges. So (N3, N2) may be replaced by (N3, N1) to produce a better graph (fig. 3c). This process can be done recursively.

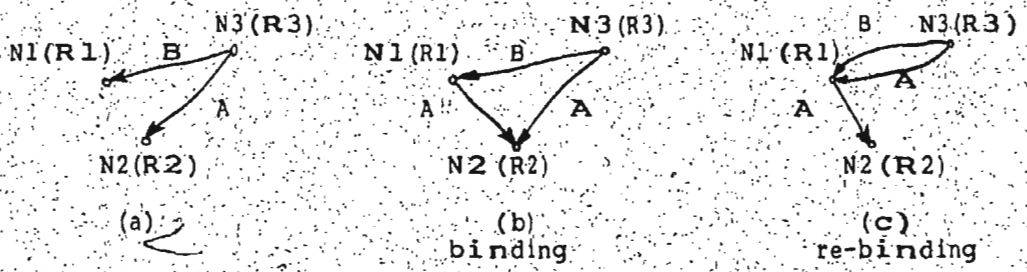


Fig. 3: An example of re-binding

Adding an edge to the strategy graph with one of the existing nodes as the predecessor node is called forking. The whole concept of forking and its application to optimization can be developed in exactly the same way as binding. In particular:

(i) The principle of "late" forking means that a semi-join should be branched out ("forked") from a node as late as possible, to take advantage of the lower cost of the most reduced occurrence of the relation on the left side of the semi-join.

(ii) Given a node N, A_SUC(N) is a set of nodes defined recursively as follows: (1) N is in A_SUC(N) and (2) if V(R) is in A_SUC(N), then all later occurrences of R after V and all successors of V which are A-connected with this occurrence of R, belong to A_SUC(N). A fork node of A_SUC(N) is a node in A_SUC(N) such that no successors of this node are in A_SUC(N). Again, if V is a fork node, A_SUC(V) consists of only V itself.

(iii) The analogues of Lemma 4 and Lemma 5 are:

Lemma 6: Let N1(S) be an occurrence of some relation S in the strategy graph. Then N1(S)[A] $\subseteq$ N(R)[A] if and only if N1 is in A_SUC(N).

Lemma 7: Suppose a semi-join (R1, R2) with join attribute A is to be forked at node N3(R3). Then forking the semi-join at node N4(R4) instead guarantees non-increasing cost if and only if N4(R4) is in A_SUC(N3).

(iv) Re-forking an existing edge is to relocate the position of the predecessor node of the edge so that its new position is a fork node of the old position. A situation where re-forking occurs is described as follows. Given edges (N1(R1), N2(R2)), and (N3(R3), N2(R2)) with labels A and B respectively, let (R1, R3) with join attribute A be the semi-join to be added. Since forking at N2(R2) would create a cycle, (N1(R1), N3(R3)) is the edge to be inserted. We can then replace (N1(R1), N2(R2)) with (N3(R3), N2(R2)) since N2 will now be a fork node of A_SUC(N1) (fig. 4).

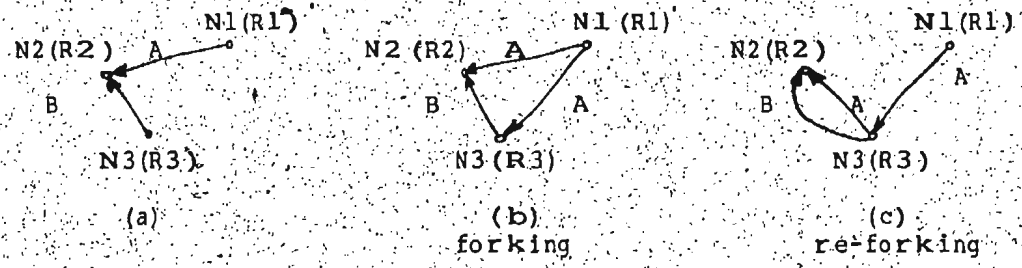


Fig. 4: An example of re-forking

There may be conflict between the principles of early binding and late forking. For example, suppose (R1, R2) is

the semi-join under consideration, and there are already occurrences of R1 and R2 in the strategy graph. According to these principles of early binding and late forking, we select N1 and N2, the last and the first occurrences of R1 and R2 respectively. If N1 is a successor of N2, we cannot have (N1(R1), N2(R2)) to represent (R1, R2), as it would then create a cycle. Under such a circumstance, we have to fork from N1 to an occurrence of R2 which is either an existing node not preceding N1 or failing that, a new node. To fork from a node containing an earlier occurrence of R1 to N2 could increase the cost of the semi-join as in P because any earlier occurrence of R1 is greater than the one in N1. Besides, if we choose a bind node first and then search for a suitable fork node, there might not be one available for the particular bind node chosen. Thus we adopt the policy of first choosing the fork node and then the bind node which either is parallel to or succeeds the fork node. Let us now apply this policy on the example above. The process of adding a semi-join proceeds in three stages:

(i) locate N1, the last occurrence of R1 and then N2, the earliest occurrence of R2 which is not a predecessor of N1. There are two possibilities:

(ii) if N2 does not exist, create a new occurrence of R2, and add an appropiate edge; or

(iii) if N2 does exist, locate the fork node of N1 and then a bind node of N2 such that no cycle is created when the

edge connecting the two is inserted.

Note that re-binding and/or re-forking are similar to insertion. The only difference is that for the former, stages (i) and (ii) are not necessary, because these conditions have been observed when the edge was first inserted. The fine-tuning stage (iii) is identical. Thus inserting a semi-join is equivalent to adding edge between N1 and N2 as in the first stage, and then re-binding (or re-forking) it.

We can now state the algorithm to construct a strategy graph. Like Algorithm 1, the input to this algorithm is a semi-join program P and the output is a strategy graph.

Algorithm 2:

DO i= 1 to n (n being the total number of semi-joins in the program P);
FOR ith semi-join (R1, R2) in P with join attribute A:

1. IF both R1 and R2 are new, THEN DO:
   (i) create new nodes N1 and N2 for R1 and R2 respectively
   (ii) add a separate connected component with one edge (i.e.(N1(R1), N2(R2))) to the graph

2. IF R1 is new but R2 is not, THEN for N2 being the first occurrence of R2, DO:
   (i) create a new node N1 for R1
   (ii) select a bind node, N3(R3) of A_PRE(N2)
   (iii) add an edge (N1(R1), N3(R3)) to the graph

3. IF R2 is new but R1 is not, THEN for N1 being the last occurrence of R1, DO:
   (i) create a new node N2 for R2.
   (ii) select a fork node N3(R3) of A_SUC(N1)
   (iii) add an edge (N3(R3), N2(R2)) to the graph

4. IF neither R1 nor R2 is new, THEN for N1 being the last occurrence of R1, DO:
   4.1. IF there is at least one occurrence of R2 which is not a predecessor of N1, THEN for N2 being the first such occurrence, DO:
      4.1.1. IF there is an occurrence of R2 in A_SUC(N1), THEN DO:
         (i) discard the semi-join (R1, R2) as it is redundant
      4.1.2. IF condition 4.1.1. is not true, THEN DO:
         (i) add an edge (N1(R1), N2(R2)) to the graph
         (ii) CALL REPLACE(N1(R1), N2(R2))
   4.2. IF every occurrence of R2 is a predecessor of N1, THEN for N6(R2) being the last occurrence of R2, DO:
      (i) create a new node N4 for R2
      (ii) select a fork node N3(R3) in A_SUC(N1)
      (iii) add an edge (N3(R3), N4(R2))
      (iv) replace every edge (N6, Ni), Ni being parallel to N4, with (N4, Ni)

Procedure REPLACE ((N1(R1), N2(R2)) with attribute A)
(i) delete (N1(R1), N2(R2))
(ii) select a fork node N3(R3) of the subset of A_SUC(N1) parallel to or preceding N2
(iii) select a bind node N4(R4) of the subset of A_PRE(N2) parallel to or succeeding N3
(iv) add an edge (N3(R3), N4(R4)) to the graph
(v) FOR every immediate successor N5(R5) of N3, N5 ≠ N4, DO:
   1. If Label(N3, N5) = A, THEN DO:
      (i) CALL REPLACE (N3(R3), N5(R5))
(vi) FOR every immediate predecessor N6(R6) of N4, N6 ≠ N3, DO:
   2. IF Label(N6, N4) = A, THEN DO:
      (i) CALL REPLACE (N6(R6), N4(R4))

Appendix A shows a sample semi-join program. It also shows the strategy graph which represents a semi-join program with the same cost/benefit, and another strategy graph as the output of the above algorithm based on the example as the input.

Algorithm 2 makes an implicit assumption that it is able to distinguish the first and last occurrences of a relation. Lemma 8 shows it is always possible. Lemmas 8 and 9 together prove that the graph produced by Algorithm 2 is indeed a strategy graph. Lemma 10 states that the semi-join programs as represented by the strategy graph satisfy the conditions c.4-c.7 in Chapter 4.

Lemma 8: All occurrences of a relation are in predecessor-successor relationship with each other and the occurrence of a relation is a subset of all previous occurrences.

Proof: The only condition under which a new occurrence of an existing relation can be created is condition 4.2 in Algorithm 2, which makes that occurrence a successor of the last existing occurrence. Q.E.D.

Lemma 9: The strategy graph constructed by Algorithm 2 contains no cycles.

Proof: Initially we have a null graph and the lemma is trivially true. The lemma is obviously true if every time an edge is added to the graph, no cycle is created. Clearly, the Lemma is true when one end node of the new edge is new. Thus we consider only those edges which replace the existing ones; that is, those edges inserted under condition 4.2. and procedure REPLACE.

Under condition 4.2., if the replacement of (N6, Ni) by (N4, Ni) creates a cycle, then there must already be a path

from Ni to N4. But this is impossible because Ni and N4 are parallel to each other. In procedure REPLACE, by the choice of N3 and N4, N4 is either parallel to N3 or it succeeds N4. Hence the edge (N3, N4) will not create a cycle. Q.E.D.

Lemma 10: Algorithm 2 represents a lossless transformation.

Proof: Every edge corresponding to a semi-join in P is inserted according to the principles of early binding and late forking. Thus by Lemmas 5 and 7, the strategy graph constructed by Algorithm 2 corresponds to a class of semi-join programs which have non-increasing cost and non-decreasing benefit compared to P. All the semi-joins in P' are either the same as the ones in P, or legal semi-joins as defined by P. So the lemma is true. Q.E.D.

We now analyse Algorithm 2. The only complicated part of Algorithm 2 is the recursive procedure REPLACE. Thus it is important to find out how it functions. This analysis will also be helpful in discussing the optimality of Algorithm 2 in the next chapter. Two important properties as stated in Lemmas 11 and 12 are largely the outcome of this analysis, which is done in the proof process of Lemma 11.

Lemma 11: In the strategy graph constructed by Algorithm 2, there are no parallel edges which have the same label and share the same predecessor (or successor) node.

Proof: We prove by induction. The lemma is obviously true when only one semi-join is processed. Assume now that the

the lemma is true after k semi-joins have been processed. We consider the (k+1)th semi-join in P.

If the edge representing this semi-join is inserted in step 1 of Algorithm 2, it is not parallel to any existing edges. In step 2, the new edge (N1, N3) cannot be parallel to any edge with label A terminating at N3 because N3 is a bind node of A_PRE(N2). In steps 3 and 4.2, similar argument applies. It remains now to analyse procedure REPLACE.

The procedure will be recursively invoked if it is discovered that the new edge (N3, N4) shares an end node with another existing edge whose label (say A) is the same. We shall first consider the case of re-forking; that is, the edge is (N3, N5). We assume that before the edge (N3, N4) is inserted, there are no parallel edges with label A, sharing the same end node. Thus (N3, N5) will be the only edge for re-forking.

Since N3 is so chosen to be a fork node of the subset of A_SUC(N1) parallel to or preceding N2, N5, being in A_SUC(N3), must be a successor of N2. But N4 is either N2 or its predecessor. So N5 is a successor of N4. As there will be no edges with label A into N5 other than (N3, N5) itself, there has to be an edge (Ni, N5) with a different label between N4 and N5. When (N3, N5) is replaced, the fork node

N7 chosen must be either N4 or one of its successors. The bind node chosen must be N5, because with the deletion of (N3, N5), A_PRE(N5) contains only N5. (That is, no re-binding is necessary.) So the edge (N7, N5) is now the successor edge of (N3, N4). If (N7, N5) causes further re-forking, the edges to be REPLACEd must be successors of (N3, N4). Thus the entire process ends when there are no more successors of (N3, N4) which have label A and are parallel to any edge with label A. Obviously, the number of iterations involved is no more than the number of successors of (N3, N4).

A similar argument applies to the re-binding process. Thus no parallel edges with the same label can be allowed in procedure REPLACE. So the lemma is true by induction. Q.E.D.

Lemma 12: In Algorithm 2, no re-binding is involved in a re-forking process and vice versa.

Proof: The lemma follows from the analysis of the procedure REPLACE in the proof of Lemma 11. Q.E.D.

Incidentally, Lemma 11 has an interesting analogue in [Hevner & Yao, 79] in which an optimal strategy is presented for processing "simple" queries, i.e. queries with one single join attribute for all the join clauses and the same attribute on the attribute list. It is proved that the optimal semi-join program is represented by a straight-line

strategy graph. The relations are ordered in ascending sequence according to the cardinalities of the join attribute. Given a semi-join program containing semi-joins with the same attribute, a straight-line strategy graph will be the outcome of Algorithm 2, according to Lemma 11. However, the corresponding semi-join program may not be optimal, since there is no consideration given to relation cardinalities in this work.

Finally, we perform the timing analysis of Algorithm 2.

Theorem 2: The running time of Algorithm 2 is O(n**3), where n is the number of semi-joins in the program.

Proof: For each semi-join in P, searching for specific occurrences of relations and locating bind/fork nodes will take O(n) operations each. If an insertion of an edge invokes procedure REPLACE, then either re-binding or re-forking is performed, by Lemma 12. Following the analysis of procedure REPLACE in the proof of lemma 11, one can see that the recursive procedure will run at most 2k times, k being the number of edges with label A at that time. Therefore the whole algorithm runs in O(n**3). Q.E.D.

Chapter 6

OPTIMALITY DISCUSSION

The strategy graph constructed by Algorithm 2 represents a class of semi-join programs which have the same costs and benefits. The algorithm however can produce different strategy graphs if different bind nodes and fork nodes are chosen. It is not possible to compare these strategy graphs because by r.1 in Chapter 4, we have no information about the relation cardinalities etc. We can however discuss whether the improvement over the original semi-join program is optimal, that is, whether one can still improve on the strategy graph constructed by algorithm 2 by another lossless transformation. We define an optimal strategy graph to be one such that any alteration of the graph will not produce another graph which has non-increasing cost and non-decreasing benefit over the existing graph. Thus we have to prove the strategy graph constructed by Algorithm 2 is optimal in the sense we have just defined. The following two lemmas will lead to this proof.

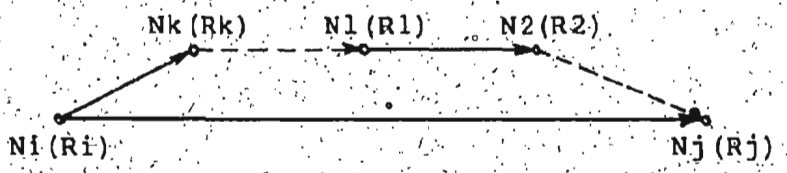
Lemma 13: The strategy graph constructed by Algorithm 2 contains no redundant edges.

Proof: We prove this lemma by induction. Initially we have a null graph. Suppose at the ith step of Algorithm 2, that is, when the ith semi-join (R1, R2) in P is the input semi-join being considered, the existing strategy graph contains no

redundant edges. We shall show that after ith step, there are no redundant edges in the graph.

Let us first show that the input semi-join, or the edge representing it, is redundant if and only if condition 4.1.1 in the algorithm is true. This is so if and only if R2[A] $\subseteq$ R1[A] before the semi-join. By lemma 7, this is true iff the occurrence of R2 is already in A_SUC(N1), where N1 is the latest occurrence of R1, i.e., condition 4.1.1 is true.

Next we show that insertion in the graph of a non-redundant input semi-join in a graph represented by an edge (N1(R1), N2(R2)) will not cause any existing edges to be redundant. Suppose there is such an edge (Ni(Ri), Nj(Rj)) with label A. This implies by Lemma 7, that with the insertion of (N1, N2), Nj(Rj) is in A_SUC(Ni) without taking into account of the edge (Ni, Nj). This is possible if and only if N1(R1) is in A_SUC(Ni), because otherwise the presence (or absence) of (N1, N2) could not have affected Nj being a node in A_SUC(Ni). But condition 4.2 in Algorithm 2 guarantees that every edge starts from the latest occurrence of a relation. Thus there cannot be another occurrence of Ri between Ni(Ri) and N1(R1). Thus if N1 is to be reduced by Ni with join attribute A, there must be an edge with label A from Ni to Nk(Rk), a successor of Ni, as shown below:



This means there are two parallel edges starting from Ni with the same label. But this cannot happen according to Lemma 11. Q.E.D.

Lemma 14: The strategy graph constructed by Algorithm 2 can not be improved by means of binding and/or forking.

Proof: We first show that if an edge (Ni, Nj) with label A is such that (i) Ni is the only node in A_SUC(Ni) parallel to or preceding Nj, and (ii) Nj is the only node in A_PRE(Nj) parallel to or succeeding Ni, then (Ni, Nj) cannot be re-forked or re-bound.

In the case of re-binding, if (Ni, Nj) is replaced by another edge (Ni, Nk), k≠j, then Nk must be in A_PRE(Nj). This means Nk must be a predecessor of Ni, so that the insertion of (Ni, Nk) will create a cycle. A similar argument applies to the case of re-forking.

Suppose now all edges in the graph at the ith step of Algorithm 2 satisfy (i) and (ii). Let us show that the same is also true at the next step.

If the next semi-join causes the new edge (N1, N3) to be inserted under condition 2, the edge must satisfy (i) and

(ii). There cannot be another node N4 which has a directed edge with label A to N3, since otherwise N4 and not N3 would be a bind node of A_PRE(N2), a contradiction. Thus the insertion of this edge will not cause any existing edge to violate (i) and (ii). Thus the lemma is true if the addition is done under condition 2. A similar argument applies to condition 3.

The insertion of (N3, N4) under condition 4.2 is very similar to the one under condition 3, so that (N3, N4) must satisfy (i) and (ii). Unlike condition 3, however, N4 is another occurrence of an existing relation R2. Thus the insertion of (N3, N4) might cause other edges not to satsify (i) and (ii). For instance, for any join attribute X, X_SUC(N6), where N6 is the last occurrence of R2 before N4, now contains N4 as well. Let us now determine the nodes whose status as bind nodes and/or fork nodes will be affected by the insertion of (N3, N4). First of all, no bind node need be updated as the new node added is not a predecessor to any existing node and hence no earlier binding is required. By the definition of fork node, addition of the node N4 will affect only N3 and N6 as fork nodes. But other than (N3, N4) there cannot be any edges with label A and having N3 as predecessor node, by lemma 11. So, only those edges that fork from N6 will be affected. Thus all edges, irrespective of their labels which start

from N6 would have to start from N4, except those edges which are predecessors of (N3, N4). This is precisely what (iv) under condition 4.2 does.

In procedure REPLACE, it is easy to see that the new edge (N3, N4) must satisfy (i) and (ii). The only edges that might be affected by this insertion are those which have the same label (i.e. A) and have either N3 as the fork node (N4 is now in A_SUC(N3)) or N4 as the bind node (N3 is now in A_PRE(N4)). They are edges which are parallel to (N3, N4) and share one of its end nodes, and their replacements are accomplished by (v) and (vi) of the procedure. Q.E.D.

Theorem 3 The strategy graph constructed by Algorithm 2 is optimal.

Proof: Alterations of a graph include: (i) additions of new edges, (ii) deletions of existing edges and (iii) re-positionings of the existing edge. The first type of alteration is not applicable because of c.4 in Chapter 4. To delete an edge, the edge has to be redundant, i.e. the edge provides no benefit. Lemma 13 above shows the strategy graph constructed by Algorithm 2 has no redundant edges. Re-positioning the predecessor node of an edge must not increase the cost of the edge, and as such, it amounts to re-forking the edge. Likewise, re-positioning the successor node of an edge amounts to re-binding an edge. Lemma 14 shows that there can be no re-binding or re-forking of any

edge of a strategy graph constructed by Algorithm 2. Q.E.D.

Chapter 7

SUMMARY AND CONCLUSIONS

We have constructed a graph representation of a semi-join program. Two algorithms have been presented to transform this program into graphs. The first algorithm produces a graph which has the same cost/benefit as the input semi-join program. The significance of this algorithm is not only in demonstrating the validity of the graph representation, but also in providing insight into the optimization of the graph performed by Algorithm 2. Algorithm 2 transforms any given semi-join program into a 'better' one. The improvement has been proved to be optimal in the sense that no further improvement can be achieved without the chance of increasing the cost and/or decreasing the benefit. Thus the improvement is absolute, non-stochastic and independent of any probabilistic estimation method. Finally, it is hoped that the results reported in this thesis will contribute to a better understanding of distributed query processing using semi-joins.

REFERENCES

[Bernstein & Chiu, 81]
Bernstein, P.A. and D.W. Chiu, "Using Semi-Joins to Solve Relational Queries", Journal of ACM, Vol. 28, No. 1, January 1981, pp. 25-40.

[Bernstein & Goodman, 79]
Bernstein, P.A. and N. Goodman, "Full Reducers for Relational Queries using Multi-attribute Semi-Joins", Proc. 1979 NBS Symp. on Computer Network

[Chiu & Ho, 80]
Chiu, D.W. and Y.C. Ho, "A Method for Interpreting Tree Queries into optimal Semi-Join Expressions", Proc. ACM SIGMOD Int. Conf. on Management of Data 1980, pp. 169-178.

[Date, 81]
Date, C.J., "Introduction to Database Systems", 3rd Ed., Addison-Welsey, 1981

[Bernstein et al., 81]
Bernstein, P.A., N. Goodman,, E. Wong, C.L. Reeve and J.B. Rothnie, "Query Processing in SDD-1: A System for Distributed Databases", ACM Trans. on Database Systems Vol. 6, No.4, 1981, pp. 612-625.

[Hevner, 79]
Hevner, A.R., "Optimization of Query Processing in Distributed Database Systems", Ph.D. Thesis, Purdue University, 1979

[Hevner & Yao, 79]
Hevner, A.R. and S.B. Yao, "Query Processing in Distributed Database Systems", IEEE Trans. on Software Engineering Vol. SE-5, No. 3, 1979, pp. 177-187

[Yu et al., 80]
Yu, C.T., K. Lam and M. Ozsoyoglu, "Distributed Query Optimization for Tree Queries", Technical Report, Department of Information Engineering, University of Illinois at Chicago Circle, July 1980.

[Yu et al., 82]
Yu, C.T., K. Lam, C.C. Chang and S.K. Chang, "A Promising Approach to Distributed Query Processing", Proc. of the Sixth Berkeley Workshop on Distributed Data Managment and Computer Networks, 1982, pp. 363-390.

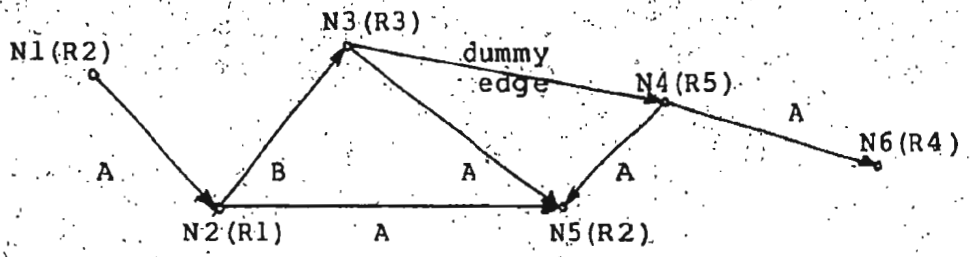
Appendix A

A SEMI-JOIN PROGRAM AND ITS ASSOCIATED STRATEGY GRAPHS

1. An example semi-join program:

   (R2, R1) - A as join attribute
   (R1, R3) - B as join attribute
   (R5, R2) - A as join attribute
   (R1, R2) - A as join attribute
   (R3, R2) - A as join attribute
   (R5, R4) - A as join attribute

2. A strategy graph representing semi-join programs with the same cost/benefit (i.e. output of Algorithm 1):



3. The strategy graph as the output of Algorithm 2:

   Events: Early binding of (R5, R2)
   Late forking of (R5, R4)
   Re-forking of the edge (N2(R1), N5(R2))