Using Semi-Joins to Solve Relational Queries

PHILIP A. BERNSTEIN AND DAH-MING W. CHIU

Harvard University, Cambridge, Massachusetts

ABSTRACT. The semi-join is a relational algebraic operation that selects a set of tuples in one relation that match one or more tuples of another relation on the joining domains. Semi-joins have been used as a basic ingredient in query processing strategies for a number of hardware and software database systems. However, not all queries can be solved entirely using semi-joins. In this paper the exact class of relational queries that can be solved using semi-joins is shown. It is also shown that queries outside of this class may not even be partially solvable using "short" semi-join programs. In addition, a linear-time membership test for this class is presented.

KEY WORDS AND PHRASES: semi-join, relational database, relational query processing
CR CATEGORIES: 3.74, 4.33, 5.25

1. *Introduction*

The utility of very high level database query languages based on the relational database model, such as QUEL [8] and SEQUEL [4], is predicated on the ability to implement these languages efficiently. Many schemes for interpreting relational query languages in centralized systems, distributed systems, and on special parallel hardware have been devised (e.g., [9, 10, 12–14]). These schemes are essentially strategies for executing a combination of projection, restriction, and join operators from relational algebra to answer the given query [7]. Two of these methods, the distributed retrieval algorithm for SDD-1 [14] and the RAP database machine [9], use a special operator that is a composed join and projection. The properties of this operator, which we call the *semi-join* and denote by $\ltimes$, are the subject of this paper.

The semi-join operator takes the join of two relations, R and S, and then projects back out on the domains of relation R.[1] That is, it retrieves those tuples in R that join with some tuple in S. Alternatively, one can think of semi-join as a generalization of restriction; it restricts R by values that appear in S's join domain.

The value of the semi-join operator is that it can reduce the amount of effort required to do a later expensive join, while the semi-join itself is often quite cheap.

[1] The join of relations $R(A, B)$ and $S(B, C)$ on domain B is the set of tuples $\{\langle a, b, c\rangle \in A \times B \times C | \langle a, b\rangle \in R$ and $\langle b, c\rangle \in S\}$. The projection of relation $T(A, B, C)$ on domains A and B is the set $\{\langle a, b\rangle \in A \times B | \exists c \in C(\langle a, b, c\rangle \in T)\}$, that is, the join of R and S projected back on the domains of R.

This work was supported in part by the National Science Foundation under Grants MCS-77-05314 and ENG-76-11824, by the Joint Services Program under Contract N00014-75-C-0648, and by the Defense Advanced Research Projects Agency of the Department of Defense, monitored by the Naval Electronic Systems Command under Contract N0039-77-C-0074.

Authors' addresses: P. A. Bernstein, Aiken Computation Laboratory, Harvard University, Cambridge, MA 02138; D. W. Chiu, Pierce Hall, Harvard University, Cambridge, MA 02138.

For example, suppose a query requires that the join of R and S be constructed. Instead of computing the join directly, one can first reduce the size of R and S by semi-joins (i.e., $R \ltimes S$ and $S \ltimes R$) and then construct the join of the resulting relations. No information is lost by the preliminary semi-joins. For the semi-joins to be performed, only the projection of the joining columns need be sent. If the size of these projections is small relative to the amount by which R and S are reduced, then the preliminary semi-joins will be profitable.

To reduce the cost of processing joins, semi-joins play a pivotal role in the query processing algorithm of SDD-1, a prototype distributed database system. The principal problem in evaluating a relational query on a distributed database is that two relations that must be joined may reside at different sites. For the join to be performed, one relation must be shipped to the site of the other. Communication is the dominant cost in a distributed database system, so minimizing the amount of data to be shipped is of prime importance [11]. Since the join must eventually be performed, one can only hope to limit communication, not eliminate it. One tactic for limiting communication is to perform first as much local processing of the query as possible; for example, all restrictions should be applied early. Another tactic is to use semi-joins. To join R and S at different sites, one can ship the projection of R on its joining column to S's site and use a semi-join to reduce S by R before shipping S to R's site. This will be a profitable tactic whenever the projection of R on its joining column is smaller than the amount by which S is reduced by the semi-join. The SDD-1 algorithm uses a heuristic hill-climbing strategy that tries to apply as many such semi-joins as are profitable.

Semi-joins are also important for the RAP machine, a hardware device tailored for relational query processing. RAP is not capable of performing joins itself and must therefore ship data to a conventional CPU to perform them. To reduce the amount of data shipped, the RAP designers have provided a hardware semi-join instruction that they recommend using to partially evaluate queries before performing joins on an external processor.

Since the database is of finite size, there are limits to how much the database can be reduced relative to a given query using semi-joins. Knowing which sequences of semi-joins can fully reduce the size of the database is important in selecting optimal retrieval strategies for RAP, SDD-1, and other systems as well. In this paper we examine the question of which semi-join sequences are efficacious. We show that relational calculus queries are naturally partitioned into two broad classes: One class can be fully evaluated using a small number of semi-joins; the other class can be neither fully evaluated using semi-joins alone nor easily reduced using a small number of semi-joins. For the one class, semi-join solution strategies are well behaved; for the other they can be dismally poor. We proceed formally by developing some basic properties of semi-joins and then showing the behavior of semi-joins with respect to these two classes of queries.

2. *Relations and Semi-Joins*

A relation is a subset of the Cartesian product of its domains. We distinguish between relation names, denoted by $\{R_1, R_2, \ldots\}$, and their corresponding relations (i.e., sets of tuples), denoted by $\{S_1, S_2, \ldots\}$. A *database* D is defined to be an ordered set of relation names $\mathbf{R}$. A *database state* is, in turn, a pair $\langle \mathbf{S}, env \rangle$, where $\mathbf{S}$ is an ordered set of relations and $env: \mathbf{R} \rightarrow \mathbf{S}$. For notational convenience we will assume that $\mathbf{R} = \{R_1, \ldots, R_n\}$, $\mathbf{S} = \{S_1, \ldots, S_n\}$, and $env(R_i) = S_i$ for $i = 1, \ldots, n$.

Note that we use the standard mathematical definition of a relation as a fixed

subset of the Cartesian product of its domains. This differs slightly from some uses in the database literature where relation means a "time-varying subset of the Cartesian product of its domains" that changes as operations are applied (e.g., [6]). Thus, in our model an operation maps one database state into another. In practice, of course, only a single copy of the database is maintained. Update operations modify the copy; retrieval operations produce a temporary database that is the subset of the real database that the query requested. We adopt our model for mathematical simplicity only and do not suggest that entire databases be created and destroyed as a consequence of each operation.

Given two database states $\mathbf{S}'$ and $\mathbf{S}''$, we define $\mathbf{S}' \cup \mathbf{S}'' = \{S_i' \cup S_i'' \mid i = 1, \ldots, n\}$ to be their componentwise union. Also, we define a partial order $\leq$ on database states. We write $\mathbf{S}' \leq \mathbf{S}''$ if $S_i' \subseteq S_i''$, $i = 1, \ldots, n$. Similarly, $\mathbf{S}' < \mathbf{S}''$ if $S_i' \subsetneq S_i''$, $i = 1, \ldots, n$.

We denote the semi-join operation by the symbol $\ltimes$. The semi-join of R_i on domain A with R_j on domain B is defined as a function from database states into database states: $\langle R_i\ {}_A\ltimes_B\ R_j\rangle(\mathbf{S}) = \mathbf{S}'$, where $S_k' = S_k$ for $k \neq i$ and $S_i' = \{t_i \in S_i \mid \exists t_j \in S_j$ such that $t_i.A = t_j.B\}$. The relational calculus notation $t.X$ means "the value of domain X of tuple t." When there is only one semi-join on R_i by R_j, we drop domain references and simply write $R_i \ltimes R_j$.

We limit ourselves to semi-joins involving single domains. Our justification is practicality. To perform a semi-join of, say, R_i on domains A and B with R_j on domains C and D requires projecting R_j on a pair of domains C and D. The size of such a projection will generally be quite large and therefore the benefit of the semi-join is unlikely to exceed its cost. For this reason, multidomain semi-joins are ignored in SDD-1 [14] and RAP [9]. Still, we do allow multidomain semi-joins in a limited context. If a pair of domains in a relation are *always* treated as a single composite domain (i.e., neither domain ever participates in a semi-join by itself), then the composite domain can be considered to be atomic and all of our results follow correctly.

LEMMA 1. *If* $S_1 = S_1' \cup S_1''$, *then*

(*a*) $\langle R_2 \ltimes R_1\rangle(\{S_1, S_2\}) = \langle R_2 \ltimes R_1\rangle(\{S_1', S_2\}) \cup \langle R_2 \ltimes R_1\rangle(\{S_1'', S_2\})$;
(*b*) $\langle R_1 \ltimes R_2\rangle(\{S_1, S_2\}) = \langle R_1 \ltimes R_2\rangle(\{S_1', S_2\}) \cup \langle R_1 \ltimes R_2\rangle(\{S_1'', S_2\})$.

PROOF. Follows directly from the definition of $\ltimes$. □

For evaluating a relational database query, in general many semi-joins are required. Hence we define a *semi-join program* to be a sequence of semi-joins, denoted $\sigma = \langle R_{i_1} \ltimes R_{i_2}, \ldots, R_{i_{2m-1}} \ltimes R_{i_{2m}}\rangle$, whose meaning is simply the composition of the semi-joins, $\sigma = \langle R_{i_1} \ltimes R_{i_2}\rangle \cdot \cdots \cdot \langle R_{i_{2m-1}} \ltimes R_{i_{2m}}\rangle$. The following lemma shows that semi-join programs obey a simple monotonicity property.

LEMMA 2. *If* $\mathbf{S}' \leq \mathbf{S}''$, *then* $\sigma(\mathbf{S}') \leq \sigma(\mathbf{S}'')$.

PROOF. We proceed by induction on the length of σ. The basis step goes as follows: Suppose $\sigma = \langle R_2 \ltimes R_1\rangle$. Then,

$$\begin{aligned}\sigma(\{S_1'', S_2''\}) &= \sigma(\{S_1', S_2''\}) \cup \sigma(\{S_1'' - S_1', S_2''\}) && \text{(by Lemma 1(a))}\\ &= \sigma(\{S_1', S_2'\}) \cup \sigma(\{S_1', S_2'' - S_2'\})\\ &\quad \cup \sigma(\{S_1'' - S_1', S_2''\}) && \text{(by Lemma 1(b))}\end{aligned}$$

Thus $\sigma(\{S_1', S_2'\}) \leq \sigma(\{S_1'', S_2''\})$. Since all relations other than S_2 remain unchanged by σ, $\sigma(\mathbf{S}') \leq \sigma(\mathbf{S}'')$.

The induction step follows similarly. □

3. *Queries*

We are interested in applying semi-joins to evaluate relational database queries. We will express our queries in a notation similar to that of relational calculus [7]. Formally, an *equi-join qualification* q is a conjunction of clauses of the form $(R_i.X = R_j.Y)$, where X and Y are domains of R_i and R_j, respectively. Each equi-join qualification q defines an equi-join query Q that produces a relation from each database state as follows:

$$Q(\mathbf{S}) = \{\langle t_1, \ldots, t_n\rangle \in S_1 \times \cdots \times S_n \mid q(t_1, \ldots, t_n)\}.$$

(Since each t_i is a tuple, $\langle t_1, \ldots, t_n\rangle$ is a tuple of tuples.) In words, $Q(\mathbf{S})$ is the subset of $S_1 \times \cdots \times S_n$ that satisfies q. We say that two queries are *equivalent*, denoted $Q_1 \equiv Q_2$, if $Q_1(\mathbf{S}) = Q_2(\mathbf{S})$ for all $\mathbf{S}$ (this is called "strong equivalence" in [2]).

Notice that equi-join queries do not include one-relation clauses of the form $(R_i.X = \text{constant})$, nor do they include target lists. One-relation clauses were excluded because they are generally evaluated using special techniques before semi-joins are applied. For example, in SDD-1 one-relation clauses correspond to local operations that do not require semi-joins [14]; in RAP one-relation clauses are evaluated in one rotational delay using the MARK operation [9]. In any case, by creating a relation containing a single constant tuple, these one-relation clauses can be considered in equi-join queries.

Target lists were excluded because the semi-join does not have the power to "project out" certain domains. A single projection operation applied to the result of an equi-join query is always sufficient to obtain the full power of target lists.[2]

Our definition of query *does* allow multidomain joins between two relations by conjoining two clauses. For example, to join R_i on domains W and X with R_j on domains Y and Z, the clauses $((R_i.W = R_j.Y) \wedge (R_i.X = R_j.Z))$ are used.

We also allow one-relation joins of the form $(R_i.A = R_i.B)$. Like simple one-variable clauses, these one-relation joins are also generally evaluated using special techniques before semi-joins are applied. However, syntactically eliminating them from our class of queries is futile, since a one-relation join can be in the closure of q even if it is not in q itself. For example, $(R_i.A = R_j.C)$ and $(R_j.C = R_i.B)$ implies $(R_i.A = R_i.B)$. So, although we will want to think of these one-relation joins as executing "for free," we still need them in our definition of query. More will be said on this issue in Section 4.

It will occasionally be convenient to examine the components of $Q(\mathbf{S})$ that come from each of $S_1, \ldots, S_n$. We introduce projection for this purpose. The notation $Q(\mathbf{S})[R_i]$ denotes $\{t_i \in S_i \mid \langle t_1, \ldots, t_i, \ldots, t_n\rangle \in Q(\mathbf{S})\}$. Similarly, $\mathbf{S}[R_i]$ denotes S_i.

4. *Solving Queries by Semi-Join Reduction*

4.1 REDUCTIONS. The main purpose of semi-joins is to *reduce* the number of tuples involved in the evaluation of a query. Let Q be a query and $\mathbf{S}$ be a database state. We define REDUCTIONS$(Q, \mathbf{S}) = \{\mathbf{S}' \mid \mathbf{S}' \leq \mathbf{S}$ and $Q(\mathbf{S}') = Q(\mathbf{S})\}$. A *full reduction* $\mathbf{S}^*$ of $\mathbf{S}$ with respect to Q is an element of REDUCTIONS$(Q, \mathbf{S})$ such that there is no $\mathbf{S}'$ in REDUCTIONS$(Q, \mathbf{S})$ with $\mathbf{S}' < \mathbf{S}^*$. We denote $\mathbf{S}^*[R_i]$ by S_i^*.

LEMMA 3. *For each Q and $\mathbf{S}$ there exists a unique $\mathbf{S}^*$ and $\mathbf{S}^* = \{Q(\mathbf{S})[R_1], \ldots, Q(\mathbf{S})[R_n]\}$.*

PROOF. Let $\mathbf{S}_{\text{red}} = \{Q(\mathbf{S})[R_1], \ldots, Q(\mathbf{S})[R_n]\}$ and let $\mathbf{S}^*$ be a full reduction. It

[2] The lack of negation, an existential quantifier, and tuple variables imply that equi-join queries are not "relationally complete" in the sense of [7].

$\mathbf{S} = \{S_1, S_2, S_3\}$.

$S_1(A\ B)$	$S_2(C\ D)$	$S_3(E\ F)$
0 1	1 2	2 3
3 4	4 5	5 0

Let Q be a query with qualification q.

$q = (R_1.B = R_2.C) \wedge (R_2.D = R_3.E) \wedge (R_3.F = R_1.A)$.

$\mathbf{S}^* = \{\varnothing, \varnothing, \varnothing\}$.

Yet for each join clause, the corresponding semi-join does not reduce at all.

FIG. 1. An example where $\mathbf{S}^*$ is not obtainable by semi-joins.

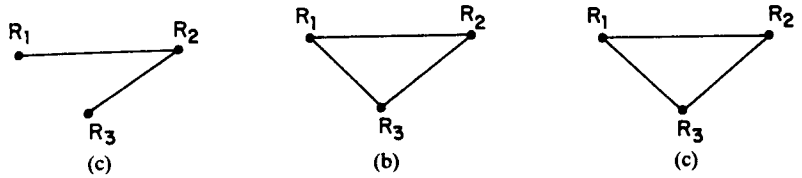


FIG. 2. (a) Query graph G_{Q_1} for query Q_1 defined by qualification $q_1 = (R_1.A = R_2.B) \wedge (R_2.B = R_3.C)$. (b) Query graph G_{Q_2} for query Q_2 defined by qualification $q_2 = (R_1.A = R_2.B) \wedge (R_2.B = R_3.C) \wedge (R_3.C = R_1.A)$. (c) Query graph G_{Q_3} for query Q_3 defined by qualification $q_3 = (R_1.A = R_2.B) \wedge (R_2.C = R_3.D) \wedge (R_3.E = R_1.F)$.

must be that $\mathbf{S}_{\text{red}} \leq \mathbf{S}^*$, for otherwise $Q(\mathbf{S}^*) \subsetneq Q(\mathbf{S})$. Since this holds for any full reduction $\mathbf{S}^*$, it follows that $\mathbf{S}_{\text{red}}$ is the unique full reduction. □

The principal issue of this paper is discovering when semi-join programs can completely resolve an equi-join query. This is equivalent to asking when a semi-join program can obtain $\mathbf{S}^*$. Unfortunately, $\mathbf{S}^*$ cannot always be obtained by semi-joins. For example, consider $\mathbf{S}$ and Q in Figure 1. In this case, semi-joins cannot reduce $\mathbf{S}$ at all with respect to Q, yet $\mathbf{S}^* = \{\varnothing, \varnothing, \varnothing\}$.

We now show a class of queries for which $\mathbf{S}^*$ can always be obtained by semi-joins. Later we will show that for any query outside this class, semi-join programs cannot produce $\mathbf{S}^*$ in general.

4.2 TREE QUERIES. Given a query Q with qualification q we define its corresponding *query graph* $G_Q(V_Q, E_Q)$ as (see Figure 2):

V_Q = set of all relation names referenced by q;
$E_Q = \{(i, j) \mid i \neq j$ and some clause of q references both R_i and $R_j\}$.

Since we allow more than one join between two relations, E_Q is a multiset (i.e., may contain duplicates), so G_Q is a multigraph.

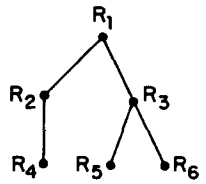
We partition the set of all equi-join queries into two classes. We call a query a *tree query* either if its query graph is a tree or if it is equivalent to a query whose query graph is a tree. We denote the set of all tree queries by TQ and all other equi-join queries by CQ (for *cyclic queries*).

Consider the set of queries in Figure 2. Since G_{Q_1} is a tree, $Q_1 \in TQ$. Although G_{Q_2} has a cycle, $Q_1 \equiv Q_2$, so that $Q_2 \in TQ$ also. The cycle in G_{Q_3} is inherent; Q_3 is not equivalent to any query whose graph is a tree, so $Q_3 \in CQ$. A simple efficient algorithm that tests whether a query is in TQ appears in Section 5.

In the sequel we will only consider equi-join queries whose query graphs are connected. A query whose query graph is disconnected produces a result that is the

Let Q be a query with qualification q.

$q = (R_1.A = R_2.B) \wedge (R_1.C = R_3.D) \wedge (R_2.E = R_4.F)$
$\wedge (R_3.G = R_5.H) \wedge (R_3.J = R_6.K)$.



$\sigma_{Q,1} = \langle R_2 \ltimes R_4, R_3 \ltimes R_5, R_3 \ltimes R_6, R_1 \ltimes R_2, R_1 \ltimes R_3 \rangle$.

FIG. 3. An example reducing semi-join program.

Cartesian product of database substates produced by each connected component. That is, since these connected components are not joined in any way, there can be no interaction between them with respect to semi-joins. Hence, there is no loss of generality in treating the components separately.

We can now state our main result.

THEOREM 1. *For any $Q \in TQ$, there is a semi-join program σ_Q such that for all* S, $\sigma_Q(\mathbf{S}) = \mathbf{S}^*$ *and* $|\sigma_Q| = 2n - 2$, *where n is the number of relations referenced by Q. Furthermore, for each i, $1 \le i \le n$, S_i^* can be obtained from* S *in $n - 1$ semi-joins.*

Theorem 1 says that there is always a short semi-join program that produces $\mathbf{S}^*$ for $Q \in TQ$. To prove Theorem 1, we construct a semi-join program that obtains S_i^* from S.

We begin by briefly reconsidering one-relation clauses of the form $(R_i.A = R_i.B)$. These clauses, we will assume, are applied to the database state before any semi-joins are executed. They are assumed to be free and do not appear in the semi-join programs that we construct to solve Q.

Let $Q \in TQ$ be a query whose graph is a tree, and let $i \in V_Q$ be a relation referenced by Q.[3] We use G_Q as a control structure to guide our semi-join program that evaluates Q. In G_Q, choose i as the root of the tree and suppose i has $m - 1$ children, $m \ge 1$. For simplicity, we rename the relations so that the root of the tree is 1 (formerly i) and its children are $2, \ldots, m$. Each of these children is the root of a query subtree, and therefore they define the set of queries $Q_2, \ldots, Q_m$. We now recursively define the *reducing semi-join program for* 1 *in* Q, denoted $\sigma_{Q,1}$, as follows:

(i) if $m = 1$, then $\sigma_{Q,1} = \langle\ \rangle$, the empty program;
(ii) else $\sigma_{Q,1} = \langle \sigma_{Q_2,2}, \ldots, \sigma_{Q_m,m}, R_1 \ltimes R_2, R_1 \ltimes R_3, \ldots, R_1 \ltimes R_m \rangle$.

To produce $\sigma_{Q,i}$, rename the relations with their original indices.

In words, $\sigma_{Q,1}$ executes one semi-join per edge in G_Q, in a breadth-first leaf-to-root order. Equivalently, reducing semi-join programs can be produced mechanically from queries using the detachment algorithm of Wong and Youssefi [13]. An example reducing semi-join program appears in Figure 3.

The following lemma shows that $\sigma_{Q,i}$ is the program we are looking for to prove Theorem 1.

LEMMA 4. *Let Q be a query with qualification q whose query graph is a tree. Choose $i \in V_Q$ as the root of the tree. Then $\sigma_{Q,i}(\mathbf{S})[R_i] = S_i^* = Q(\mathbf{S})[R_i]$. That is, $\sigma_{Q,i}$ fully reduces S_i with respect to Q.*

[3] For notational convenience, we shall use i in place of R_i.

PROOF. The proof is by induction on the height of the tree G_Q (the *height* of a tree is the length of the longest path from the root to a leaf). Choose $i \in V_Q$ as the root of the tree.

Basis Step. Suppose the height of G_Q is 0. Since no join clauses on one variable are possible, q is empty. Hence $\mathbf{S} = \mathbf{S}^*$. Since $\sigma_{Q,i}$ is empty, $\sigma_{Q,i}(\mathbf{S}) = \mathbf{S}^*$.

Induction Step. Suppose the lemma is true for all queries whose query graph is a tree of height $<p$. We show it to be true if the height of G_Q equals p. As before, let us rename the relations so that i becomes 1 and i's children are $2, \ldots, m$, $m > 1$. Let $Q_2, \ldots, Q_m$ be the queries, with qualifications $q_2, \ldots, q_m$, defined by the subtrees (of G_Q) rooted by each child. Finally, let c_j, $2 \le j \le m$, be the join clause linking relation j to relation 1, and let $q' = c_2 \wedge \cdots \wedge c_m$.

From the definition of query, we have

$$\langle t_1, \ldots, t_n \rangle \in Q(\mathbf{S}) \quad \text{iff} \quad t_j \in S_j \ (1 \le j \le n) \quad \text{and} \quad q(t_1, \ldots, t_n).$$

It immediately follows that $t_1 \in Q(\mathbf{S})[R_1]$ iff

$$\exists t_j \in S_j \ (2 \le j \le n) \quad \text{such that} \quad q(t_1, \ldots, t_n). \tag{1}$$

Expanding q into its component clauses, (1) holds iff

$$\begin{aligned} &\exists t_j \in S_j \ (2 \le j \le m) \quad \text{such that} \\ &\quad (q'(t_1, \ldots, t_m) \quad \text{and} \\ &\quad \exists t_k \in S_k \ (m < k \le n) \text{ such that } (q_2(t_1, \ldots, t_n) \wedge \cdots \wedge q_m(t_1, \ldots, t_n))). \end{aligned} \tag{2}$$

Each q_j, $2 \le j \le m$, ranges over a set of relations that is disjoint from every other q_k, $k \ne j$. Thus, since each q_j is evaluated by Q_j,

$$\begin{aligned} &\exists t_j \in S_j \ (2 \le j \le m) \quad \exists t_k \in S_k \ (m < k \le n) \quad \text{such that} \\ &\quad (q_2(t_1, \ldots, t_n) \wedge \cdots \wedge q_m(t_1, \ldots, t_n)) \text{ iff } \exists t_j \in S_j \ (2 \le j \le m) \ t_j \in Q_j(\mathbf{S})[R_j]. \end{aligned} \tag{3}$$

Since each G_{Q_j}, $2 \le j \le m$, is a tree of height $<p$, by the induction hypothesis we have that $Q_j(\mathbf{S})[R_j] = \sigma_{Q_j,j}(\mathbf{S})[R_j]$. Substituting this into (3) and combining (3) with (2), we find that (1) holds iff

$$\exists t_j \in S_j \ (2 \le j \le m) \quad \text{such that} \quad (q'(t_1, \ldots, t_m) \text{ and } t_j \in \sigma_{Q_j,j}(\mathbf{S})[R_j]). \tag{4}$$

Let $S_j^+ = \sigma_{Q_j,j}(\mathbf{S})[R_j]$. Each clause of q' can be evaluated by a semi-join, so (4) holds iff

$$\begin{aligned} &\exists t_j \in S_j \ (2 \le j \le m) \quad \text{such that} \\ &\quad \langle t_1, \ldots, t_m \rangle \in \langle R_1 \ltimes R_2, \ldots, R_1 \ltimes R_m \rangle (\{S_1, S_2^+, \ldots, S_m^+\}). \end{aligned} \tag{5}$$

But $\sigma_{Q,1}(\mathbf{S})[R_1, \ldots, R_m]$ exactly calculates $\langle R_1 \ltimes R_2, \ldots, R_1 \ltimes R_m \rangle (\{S_1, S_2^+, \ldots, S_m^+\})$, so $t_1 \in Q(\mathbf{S})[R_1]$ iff $\exists t_j \in S_j$, $2 \le j \le m$, such that $\langle t_1, \ldots, t_m \rangle \in \sigma_{Q,1}(\mathbf{S})[R_1, \ldots, R_m]$. It immediately follows that $Q(\mathbf{S})[R_1] = \sigma_{Q,1}(\mathbf{S})[R_1]$, as desired. □

Note that $\sigma_{Q,i}$ produces S_i^* but not S_j^*, $j \ne i$. That is, only relation S_i is fully reduced by $\sigma_{Q,i}$. Intuitively, the reason is that each relation (except for S_i) is only reduced by a proper subset of the clauses in the query. More specifically, each relation S_j is reduced by clauses in the subtree of Q rooted at j (considering i as the root of Q's tree). To produce $\mathbf{S}^*$, we first perform $\sigma_{Q,i}$ and then apply semi-joins from the root i down the tree toward the leaves. The end effect is that each relation is reduced by every clause. Proceeding formally, the program we want is $\sigma_Q = \langle \sigma_{Q,i}, \sigma_{Q,i}^{-1} \rangle$, where $\sigma_{Q,i}^{-1}$ is defined (with the usual renaming of relations) as follows:

(i) if $m = 1$ then $\sigma_{Q,1}^{-1} = \langle\ \rangle$, the empty program;
(ii) else $\sigma_{Q,1}^{-1} = \langle R_2 \ltimes R_1, \ldots, R_m \ltimes R_1, \sigma_{Q_2,2}^{-1}, \ldots, \sigma_{Q_m,m}^{-1} \rangle$,

where 2, ..., m are the children of 1 as before. The principal property of σ_Q is that for each $j \in V_Q$, $\sigma_{Q,j}$ is *embedded* in σ_Q. That is, for each j, $\sigma_{Q,j}$ can be obtained from σ_Q simply by excising certain semi-joins from σ_Q. Clearly for all S, $\sigma_Q(\mathbf{S}) \leq \sigma_{Q,j}(\mathbf{S})$. The proof of Theorem 1 now follows directly.

PROOF OF THEOREM 1. Let $Q \in TQ$ and assume G_Q is a tree. Let $\mathbf{S}' = \sigma_Q(\mathbf{S})$. Since for each $j \in V_Q$, $\sigma_{Q,j}$ is embedded in σ_Q, it follows from Lemma 4 that $\mathbf{S}' = \mathbf{S}^*$. $\sigma_{Q,j}$ contains $n - 1$ semi-joins and σ_Q contains $2n - 2$ semi-joins.

If G_Q is not a tree, then since $Q \in TQ$, there is an equivalent query Q' whose graph is a tree. The proof goes as before, using Q' and the fact that $Q'(\mathbf{S}) = Q(\mathbf{S})$, by definition of equivalence. □

4.3 CYCLIC QUERIES. Queries in CQ are very badly behaved with respect to semi-joins compared to those in TQ. Not only are semi-joins incapable of obtaining $\mathbf{S}^*$ in general, but even the best possible reduction of S may only be obtainable by a very long semi-join program.

Let Q be a query with qualification q. We define PROGRAMS(Q) to be the set of all semi-join programs that only perform semi-joins that correspond to clauses of q. Let S be a database state. We define $\ltimes$-REDUCTIONS(Q, S) = $\{\mathbf{S}' \mid \mathbf{S}' \leq \mathbf{S}, Q(\mathbf{S}') = Q(\mathbf{S})$, and $\exists \sigma \in$ PROGRAMS(Q) such that $\sigma(\mathbf{S}) = \mathbf{S}'\}$. In words, $\ltimes$-REDUCTIONS(Q, S) is the set of all database states that can be obtained from S by semi-join programs that correspond to Q. From the definition it immediately follows that $\ltimes$-REDUCTIONS(Q, S) $\subseteq$ REDUCTIONS(Q, S). A *full* $\ltimes$*-reduction* $\mathbf{S}^\ltimes$ of S with respect to Q is an element of $\ltimes$-REDUCTIONS(Q, S) such that there is no S' in $\ltimes$-REDUCTIONS(Q, S) with $\mathbf{S}' < \mathbf{S}^\ltimes$.

LEMMA 5. *For each Q and* S *there is a unique full $\ltimes$-reduction,* $\mathbf{S}^\ltimes$.

PROOF. If $|\ltimes\text{-REDUCTIONS}(Q, \mathbf{S})| = 1$, then $\ltimes$-REDUCTIONS(Q, S) = {S} and we are done. Otherwise, let $\mathbf{S}^1$ and $\mathbf{S}^2$ be two full $\ltimes$-reductions of S with respect to Q that were produced, respectively, by σ_1 and σ_2. Since $\mathbf{S}^1$ is a full $\ltimes$-reduction, $\sigma_2(\mathbf{S}^1) = \mathbf{S}^1$. Since $\mathbf{S}^1 \leq \mathbf{S}$, by Lemma 2 $\sigma_2(\mathbf{S}^1) \leq \sigma_2(\mathbf{S}) = \mathbf{S}^2$. So $\mathbf{S}^1 \leq \mathbf{S}^2$. The symmetric argument shows that $\mathbf{S}^2 = \sigma_1(\mathbf{S}^2) \leq \sigma_1(\mathbf{S}) = \mathbf{S}^1$. Hence $\mathbf{S}^1 = \mathbf{S}^2$. □

Theorem 1 says that $\mathbf{S}^* = \mathbf{S}^\ltimes$ for tree queries. However, for cyclic queries there are states S for which $\mathbf{S}^*$ is not in $\ltimes$-REDUCTIONS(Q, S). To show that $\mathbf{S}^*$ and $\mathbf{S}^\ltimes$ are not always identical for cyclic queries, we will look at a special subclass of CQ that has minimal cycles.

Let Q be a query with qualification q. The *closure* of Q, denoted Q^+, is a query whose qualification, denoted q^+, includes q and all clauses implied by q under transitivity. (E.g., if $(R_1.A_1 = R_2.A_2)$ and $(R_2.A_2 = R_3.A_3)$ are in q, then $(R_1.A_1 = R_3.A_3)$ is in q^+.) Clearly $Q^+ \equiv Q$. Also, if $Q_1 \equiv Q_2$, then $Q_1^+ = Q_2^+$. For if $Q_1^+ \neq Q_2^+$, then there is a clause in $q_1^+ - q_2^+$, say; by selecting a state S such that some tuple in the Cartesian product of the relations in S satisfies q_2^+ but does not satisfy $q_1^+ - q_2^+$, we have $Q_1(\mathbf{S}) \neq Q_2(\mathbf{S})$, or $Q_1 \not\equiv Q_2$.[4]

LEMMA 6. *$Q_1 \equiv Q_2$ iff $Q_1^+ = Q_2^+$.*

PROOF. Follows from the above argument. □

A qualification q is called *proper cyclic* if it is of the form $\wedge_{i=1}^{p}(R_i.A_{i,2} = R_{i+1}.A_{i+1,1})$, where $p > 2$, 1 is the successor of p, and all $A_{i,j}$ are distinct. A query is *proper cyclic* if its qualification is proper cyclic.

[4] This result does not hold when tuple variables are introduced; see [5].

$S_1(A_{1,1}\ A_{1,2})$		$S_2(A_{2,1}\ A_{2,2})$		...	$S_p(A_{p,1}\ A_{p,2})$	
0	1	1	2		$p-1$	p
p	$p+1$	$p+1$	$p+2$		$2p-1$	0
a_1	a_2	a_1	a_1		a_1	a_1
a_2	a_3	a_2	a_2		a_2	a_2
⋮	⋮	a_3	a_3		a_3	a_3
a_{m-1}	a_m	⋮	⋮		⋮	⋮
a_m	1	a_m	a_m		a_m	a_m

FIG. 4. The database constructed for the proof of Lemma 7.

Intuitively, a proper cyclic qualification corresponds to a nonreducible cycle. If Q_1 is proper cyclic, then $Q_1 = Q_1^\dagger$. Also, if $Q_1 \equiv Q_2$, then (by Lemma 6) $Q_1 = Q_1^\dagger = Q_2^\dagger$. But no clause of Q_1 is implied by the others, so $Q_1 = Q_2^\dagger$ implies $Q_1 = Q_2$. Hence there is no other query equivalent to Q_1.

Proper cyclic queries are badly behaved with respect to semi-joins as indicated by the following lemma.

LEMMA 7. *For any proper cyclic query Q, there exists a database state $\mathbf{S}$ such that (i) $\mathbf{S}^\kappa \neq \mathbf{S}^*$, and (ii) the fewest number of semi-joins required to obtain $\mathbf{S}^\kappa$ is $O(m)$, where m is the number of tuples of some relation in $\mathbf{S}$.*

PROOF. Let Q have qualification $q = \bigwedge_{i=1}^{p} (R_i.A_{i,2} = R_{i+1}.A_{i+1,1})$, where $p \geq 2$. For convenience we assume that all domains of all relations are integer valued. Choose some integer $m > 1$ and a set of distinct integers $\{a_1, \ldots, a_m\}$, where $a_j \geq 2p$ for $1 \leq j \leq m$. In $\mathbf{S}$ let all of the domains not referenced by q have arbitrary values, so that in constructing $\mathbf{S}$, we consider all relations to be binary. In $\mathbf{S}$ let $S_1 = \{\langle 0, 1\rangle, \langle p, p+1\rangle, \langle a_m, 1\rangle\} \cup \{\langle a_j, a_{j+1}\rangle : 1 \leq j < m\}$, let $S_p = \{\langle p-1, p\rangle, \langle 2p-1, 0\rangle\} \cup \{\langle a_j, a_j\rangle : 1 \leq j \leq m\}$, and for each k, $1 < k < p$, let $S_k = \{\langle k-1, k\rangle, \langle p+k-1, p+k\rangle\} \cup \{\langle a_j, a_j\rangle : 1 \leq j \leq m\}$ (see Figure 4).

To show that $\mathbf{S}^* \neq \mathbf{S}^\kappa$, we observe that the tuples $\{\langle k-1, k\rangle, \langle p+k-1, p+k\rangle\}$ are in each $\mathbf{S}_k^\kappa$, $1 \leq k < p$, and $\{\langle p-1, p\rangle, \langle 2p-1, 0\rangle\}$ are in S_p. Yet $\mathbf{S}^* = \{\emptyset, \ldots, \emptyset\}$, so $\mathbf{S}^* \neq \mathbf{S}^\kappa$. In addition, $\mathbf{S}$ is constructed so that only one possible semi-join can reduce the size of any relation in $\mathbf{S}$, and furthermore every database produced by a sequence of reducing semi-joins has this property. In fact, the only (nonredundant) sequence of semi-joins that can produce $\mathbf{S}^\kappa$ is

```
do m times;
    do k = 1 to p by 1;
        R_{k+1} ⋉ R_k;
    end;
end;
```

where 1 is the successor of p. This requires pm semi-joins. □

The proof of Lemma 7 actually supports a stronger statement about semi-join programs for proper cyclic queries. The constructed database (Figure 4) is designed so that each semi-join only reduces the database by one tuple. Even if one were willing to be satisfied with a reduced database larger than $\mathbf{S}^\kappa$, obtaining such a reduction would still be slow. To obtain $\mathbf{S}' \geq \mathbf{S}^\kappa$, the number of semi-joins required equals the number of tuples in $\mathbf{S}$ minus the number in $\mathbf{S}'$.

Of course, not every cyclic query is proper cyclic. However, we can strengthen Lemma 7 to cover all cyclic queries. We begin by showing that a certain type of proper cyclic qualification is embedded in every cyclic query.

A qualification q' is a *proper cyclic subqualification of* q if

(1) q' is contained in q^+,
(2) q' is proper cyclic, and
(3) for any two domains $R_i.A_i$ and $R_j.A_j$ that appear in clauses in q', if $(R_i.A_i = R_j.A_j)$ is in q^+, then $(R_i.A_i = R_j.A_j)$ is in q.

Notice that q is a proper cyclic subqualification of q iff q is proper cyclic, since (3) is irrelevant when q is proper cyclic.

Part (3) of the above definition guarantees that the cycle corresponding to q' cannot be broken up into two shorter cycles, for if $(R_i.A_i = R_j.A_j)$ were in q' but not in q', then a "path" of clauses from $R_i.A_i$ to $R_j.A_j$ in q' could be replaced by $(R_i.A_i = R_j.A_j)$, thereby shortening or destroying the cycle.

LEMMA 8. *Query Q with qualification q is in CQ iff q has a proper cyclic subqualification.*

The proof of Lemma 8 is most conveniently presented after we have developed more machinery for manipulating equivalent queries. Section 5 develops this machinery for the purpose of testing membership in TQ, so we defer the proof to the end of that section.

Since every cyclic query has a proper cyclic subquery, Lemma 7 holds at least for the proper cyclic subquery (when it is treated as a proper cyclic query). We will show that Lemma 7 actually holds for the entire query. Since by Lemma 8 every cyclic query has a proper cyclic subquery, we can use the database of Figure 4 to produce the desired effect. However, there is a technical problem here; we must assign data values to all domains outside the proper cyclic subquery. We will show that this can be handled by making the rest of the database a Cartesian product.

Let A_i be the domains of R_i. If $B \subseteq A_i$, then $S_i[B]$ denotes the *projection* of S_i on B, where $S_i[B] = \{\langle s.B_1, \ldots, s.B_m\rangle \mid s \in S_i \text{ and } B_j \in B \text{ for } 1 \le j \le m\}$. Let $\langle R_i \times R_j\rangle$ denote the Cartesian product operation. Now if $S_i = S_i[B] \times S_i[A_i - B]$, then for any $A_{i,k} \in B$ we have

$$\langle R_{i\,A_{i,k}} \ltimes R_j\rangle(\{S_i, S_j\}) = \langle R_i[B] \times R_i[A_i - B]\rangle(\langle R_i[B] \ltimes R_j\rangle(\{S_i[B], S_i[A_i - B], S_j\})).$$

In words, if S_i is a Cartesian product of two subrelations, then any semi-join on S_i can be considered as a semi-join on the database consisting of the subrelations of S_i. A Cartesian product operation can always reconstruct the database to its original form—either before or after the semi-join.

Let $q' = \Lambda_{i=1}^{p}(R_i.A_{i,2} = R_{i+1}.A_{i+1,1})$ be a proper cyclic subqualification of some query Q. We say that a database state $\mathbf{S}$ is *decomposable* with respect to q' if

(d1) for $1 \le i \le p$, $S_i = S_i[A_{i,1}, A_{i,2}] \times (\times_{k \ne 1,2} S_i[A_{i,k}])$;
(d2) for $p < i \le n$, $S_i = \times_k S_i[A_{i,k}]$;
(d3) for all i and k, $S_i[A_{i,k}] \ne \varnothing$; and
(d4) for $1 \le i \le p$ and $k = 1, 2$, $S_i[A_{i,k}] \subseteq \bigcap_{R_j.A_{j,l} \in J(A_{i,k})} S_j[A_{j,l}]$;

where $J(A_{i,k})$ is the set of domains that join with $A_{i,k}$ in q^+.

Let DOMAINS(q) be the domains referenced in qualification q, and let RELATIONS(q) be the relations referenced in qualification q. Intuitively, by making $\mathbf{S}$ decomposable with respect to q' we have made every relation not in RELATIONS(q') a Cartesian product (by (d2)); and for each relation in RELATIONS(q') we have made those domains not in DOMAINS(q') a Cartesian product (by (d1)). Furthermore, (d3) and (d4) imply that no domain not in DOMAINS(q') can reduce any

domain in DOMAINS(q') by means of semi-joins, since the former are supersets of the latter. Finally, by the above observation about semi-joins applied to Cartesian products, we can decompose **S** so that domains not in DOMAINS(q') become single-domain relations; semi-joins applied to this decomposed database have the same effect as if they were applied to **S**.

Let q' be a proper cyclic subqualification of q, and let **S** be a database state decomposable with respect to q'. We can *decompose* **S** by applying a *decomposition mapping* which maps **S** into $\mathbf{S}_d$ so that each domain not in DOMAINS(q') becomes a single-domain relation. Having decomposed **S**, we can apply a *renaming mapping* to q (and q'), which maps q (and q') into q_d (and q'_d) by mapping domain references in q and q' on **S** into corresponding domain references in q_d and q'_d on $\mathbf{S}_d$. We make two important observations regarding the decomposed database and renamed query:

(dm1) Each relation R in RELATIONS(q'_d) is a binary relation containing only the two domains of R in DOMAINS(q'). All other relations in $\mathbf{S}_d$ are unary.

(dm2) For each domain (= unary relation) $R_k.A_k$ not in DOMAINS(q'_d), either there is no domain $R_i.A_i$ in DOMAINS(q'_d) such that $(R_k.A_k = R_i.A_i)$ is in q_d^+, or there is exactly one clause in q'_d, $(R_i.A_i = R_j.A_j)$, such that $(R_i.A_i = R_k.A_k)$ and $(R_j.A_j = R_k.A_k)$ are in q_d^+.

(dm2) follows from the fact that q'_d is a proper cyclic subqualification; if there are two (or more) distinct clauses of q'_d, $(R_{i1}.A_{i1} = R_{j1}.A_{j1})$ and $(R_{i2}.A_{i2} = R_{j2}.A_{j2})$, such that $(R_k.A_k = R_{i1}.A_{i1})$ and $(R_k.A_k = R_{i2}.A_{i2})$ are in q_d^+, then part (3) of the definition of proper cyclic subqualification is violated. Thus, the clauses of q'_d partition the relations that are not in RELATIONS(q'_d) but are connected to RELATIONS(q'_d) in the query graph of Q_d.

The following lemma extends the first part of Lemma 7 to arbitrary cyclic queries. It implies that if q' is a proper cyclic subqualification of q and $\mathbf{S}^* \neq \mathbf{S}^\kappa$ with respect to q', then $\mathbf{S}^* \neq \mathbf{S}^\kappa$ with respect to q.

LEMMA 9. *Let Q be a cyclic query with qualification q, let q' be a proper cyclic subqualification of q, and let **S** be a database state decomposable with respect to q'. Let $\mathbf{S}_d$, q_d, and q'_d be the results of decomposing **S** and renaming q and q'. Then the full semi-join reduction of $\mathbf{S}_d$ with respect to q_d (denoted $\mathbf{S}^\kappa_{d,q_d}$) equals the full semi-join reduction of $\mathbf{S}_d$ with respect to q'_d (denoted $\mathbf{S}^\kappa_{d,q'_d}$) on each relation in RELATIONS(q'_d).*

PROOF. It suffices to show that no relation $\mathbf{S}^\kappa_{d,q'_d}$[RELATIONS($q'_d$)] can be reduced by a semi-join corresponding to a clause in q_d^+. However, this follows directly from (d4) and (dm2). □

To extend the remainder of Lemma 7, we introduce some additional notation. Given a query Q and a database state **S**, we define $M(Q, \mathbf{S})$ to be the length of the shortest semi-join program to produce a full semi-join reduction of **S** with respect to Q; that is,

$$M(Q, \mathbf{S}) = \min_{\substack{\sigma \in \text{PROGRAMS}(Q) \\ \sigma(\mathbf{S}) = \mathbf{S}^\kappa}} |\sigma|.$$

LEMMA 10. *Let Q be a cyclic query with qualification q, let q' be a proper cyclic subqualification of q, and let **S** be a database state decomposable with respect to q'. Let $\mathbf{S}_d$, q_d, and q'_d be the results of decomposing **S** and renaming q and q'. Then $M(Q'_d, \mathbf{S}_d) \leq M(Q_d, \mathbf{S}_d)$.*

PROOF. Let σ be a semi-join program of length $M(Q_d, \mathbf{S}_d)$ such that $\sigma(\mathbf{S}_d) = \mathbf{S}^{\kappa}_{d,q_d}$. To prove the lemma, we will transform σ into σ' such that $M(Q'_d, \mathbf{S}_d) \leq |\sigma'| \leq |\sigma|$ and $\sigma'(Q'_d, \mathbf{S}_d) = S^{\kappa}_{d,q'_d}$. Begin by finding the first semi-join in σ, say $\langle R_{i\,A_i} \kappa_{A_j} R_j \rangle$, that reduces a relation in RELATIONS(q'_d). If ($R_i.A_i = R_j.A_j$) is in q'_d, then leave the semi-join as is. Otherwise, remove $\langle R_{i\,A_i} \kappa_{A_j} R_j \rangle$ from σ. By (d4) and (dm2), $\langle R_{i\,A_i} \kappa_{A_j} R_j \rangle$ could only reduce $S_{d,i}$ if there were a chain of semi-joins preceding it of the form $\langle R_{j1\,A_{j1}} \kappa_{A_k} R_k \rangle$, $\langle R_{j2\,A_{j2}} \kappa_{A_{j1}} R_{j1} \rangle$, ..., $R_{j\,A_j} \kappa_{A_{jm}} R_{jm} \rangle$, where ($R_k.A_k = R_i.A_i$) is the unique clause in q'_d referencing $R_i.A_i$. If the chain exists, remove it from σ and insert $\langle R_{i\,A_i} \kappa_{A_k} R_k \rangle$ in σ where $\langle R_{i\,A_i} \kappa_{A_j} R_j \rangle$ used to be; this has exactly the same effect on R_i as the chain did. Now find the second such semi-join, perform the replacement if appropriate, etc., until all semi-joins reducing relations in RELATIONS(q'_d) have been examined and/or replaced. Now all semi-joins in σ that reduce relations in RELATIONS(q'_d) correspond to clauses of q'_d. So remove from σ all semi-joins that do not reference relations in RELATIONS(q'_d), since none of these semi-joins can affect relations in RELATIONS (q'_d). The resulting semi-join program σ' is no longer than σ, and it produces the same effect on relations in RELATIONS(q'_d) as σ. Since $\sigma(Q_d, \mathbf{S}_d)$ [RELATIONS(q'_d)] = $\mathbf{S}^{\kappa}_{d,q'_d}$ by Lemma 9, $\sigma'(Q'_d, \mathbf{S}_d) = \mathbf{S}^{\kappa}_{d,q'_d}$ as desired. □

We can now extend Lemma 7 to all cyclic queries.

THEOREM 2. *For any query $Q \in CQ$, there exists a database state* S *such that* $\mathbf{S}^{\kappa} \neq \mathbf{S}^{*}$ *and the fewest number of semi-joins required to obtain* $\mathbf{S}^{\kappa}$ *is* $O(m)$, *where* m *is the number of tuples of some relation in* S.

PROOF. By Lemma 8, q has a proper cyclic subqualification q'. Let S be a database state decomposable with respect to q' such that if $\mathbf{S}_d$ is the decomposed version of S and q_d, q'_d are the renamed versions of q, q', then $\mathbf{S}_d$[RELATIONS(q'_d)] is a state satisfying Lemma 7. That is, $\mathbf{S}^{\kappa}_{d,q'_d} \neq \mathbf{S}^{*}_{d,q'_d}$ and $M(Q'_d, \mathbf{S}_d)$ is of size $O(m)$. By Lemma 9, $\mathbf{S}^{\kappa}_{d,q'_d}$[RELATIONS($q'_d$)] = $\mathbf{S}^{\kappa}_{d,q_d}$[RELATIONS($q'_d$)], so $\mathbf{S}^{\kappa}_{d,q_d} \neq \mathbf{S}^{*}_{d,q_d}$. By Lemma 10, $M(Q'_d, \mathbf{S}_d) \leq M(Q_d, \mathbf{S}_d)$, so $M(Q_d, \mathbf{S}_d)$ is at least of size $O(m)$. Since all of these results hold on S by taking the inverse of the decomposition and renaming maps, the theorem is proved. □

5. *A Fast Tree Query Membership Test*

To make use of our result that shows tree queries to be well behaved with respect to semi-join, we need a procedure that tests if a given query is in TQ. In this section we present such a test that runs in linear time. The test is constructive. If the given query is in TQ, an equivalent query is produced whose query graph is a tree.

Given a query with a cyclic query graph, we cannot immediately tell if the query is cyclic. For example, if the qualification consists of the cycle (($R_1.A = R_2.B$) and ($R_2.B = R_3.C$) and ($R_3.C = R_1.A$)), we can drop any one of the clauses; the result is a tree query that is obviously equivalent to the given query. The property of the cycle that permits us to drop one clause is that each relation participates in the cycle with only one joining domain.

There is a second situation in which a cycle can be broken without changing the meaning of the query. Consider the qualification (($R_1.A = R_2.B$) and ($R_2.B = R_3.C$) and ($R_3.C = R_1.D$)), which produces a cyclic query graph with edges {(R_1, R_2), (R_2, R_3), (R_3, R_1)}. This qualification does not fit the form of our previous example, because R_1 has two joining domains. Yet we can transform the qualification into (($R_1.A = R_2.B$) and ($R_2.B = R_3.C$) and ($R_1.A = R_1.D$)); since $R_1.A$ must equal $R_3.C$ in the first qualification, we can substitute ($R_1.A = R_1.D$) for ($R_3.C = R_1.D$). The

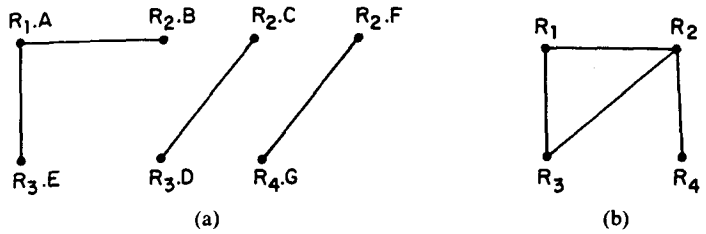


FIG. 5. A join graph and its corresponding query graph for query Q with qualification $q = ((R_1.A = R_2.B) \wedge (R_2.C = R_3.D) \wedge (R_3.E = R_1.A) \wedge (R_2.F = R_4.G))$. (a) Join graph J_Q. (b) Query graph G_Q.

resulting query graph contains edges $\{(R_1, R_2), (R_2, R_3)\}$ and is now acyclic; we have replaced an interrelation clause by an intrarelation clause that does not produce a query graph edge. Intuitively, we have replaced a join between two relations by a restriction within a single relation, thereby breaking the cycle. As we will now show formally, the transformations described by the above two examples are the only ones needed to map a tree query whose query graph is cyclic into an equivalent query whose query graph is a tree.

To perform the above transformations on clauses, we will use another graph model of a query. For a query Q with qualification q, we define the *join graph* for Q, $J_Q(V, E)$, to be a node-labeled undirected graph where

$$V = \{R_i.A \mid A \text{ is a domain of } R_i\}$$

and

$$E = \{(R_i.A, R_j.B) \mid (R_i.A = R_j.B) \text{ is a join clause in } Q\}.$$

The join graph simply represents the joins in a qualification by edges in the graph (e.g., see Figure 5). We say that a join graph J *corresponds to* a query graph G if the query represented by J has (the unique) query graph G. Unlike query graphs, join graphs are not multigraphs.

The transitive closure of J_Q, denoted J_Q^+, represents all join clauses that are logically implied by Q's qualification. A *spanning forest* of J_Q^+ is defined as a minimal subgraph of J_Q^+ whose closure is J_Q^+. A spanning forest of J_Q^+ is in some sense a minimal representation of a query. The following lemma says that to test if $Q \in TQ$ we need only look at spanning forests of J_Q^+.

LEMMA 11. *$Q \in TQ$ iff there exists a spanning forest of J_Q^+ that corresponds to an acyclic query graph.*

PROOF. Since a spanning forest of J_Q^+ has the same transitive closure as J_Q, the query represented by the spanning forest is equivalent to Q. If the spanning forest corresponds to an acyclic query graph, then Q is obviously equivalent to a query whose query graph is a tree, and hence $Q \in TQ$. If $Q \in TQ$, then it is equivalent to some Q' whose query graph is a tree. Any spanning forest of its join graph $J_{Q'}$ is a spanning forest of J_Q^+ and therefore satisfies the conditions of the lemma. □

There is a particular kind of spanning forest of J_Q^+ which is easy to compute and which has the property that $Q \in TQ$ iff such a spanning forest corresponds to an acyclic query graph. This kind of *canonical spanning forest* is constructed as follows

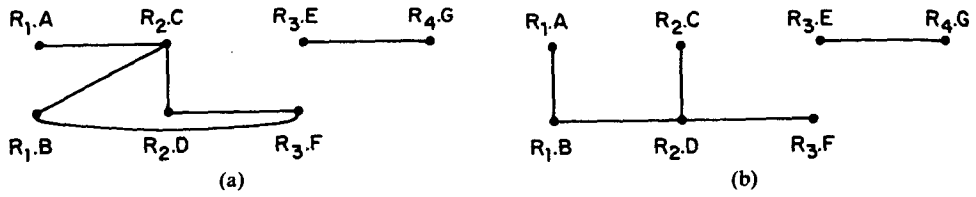


FIG. 6. A canonical spanning forest for query Q with qualification $q = ((R_1.A = R_2.C) \wedge (R_2.C = R_2.D) \wedge (R_2.C = R_1.B) \wedge (R_2.D = R_3.F) \wedge (R_1.B = R_3.F) \wedge (R_3.E = R_4.G)$. (a) Join graph J_Q (b) A canonical spanning forest of J_Q^+

(see Figure 6 for an example):

do for each connected component of J_Q;
  partition the nodes according to the relations that name them;
  for each partition, connect all the nodes in the partition by a chain;
  choose one node from each partition and connect them by a chain;
end;

Each tree in the forest spans a connected component of J_Q^+. Since there is some choice in constructing each tree, canonical spanning forests are not unique for each J_Q. However, the following lemma shows that any one of them can tell if $Q \in TQ$.

LEMMA 12. *For every query Q and for every canonical spanning forest J^C of J_Q^+, $Q \in TQ$ iff J^C corresponds to an acyclic query graph.*

PROOF. The if part follows from Lemma 11. To prove the only-if part, suppose J^C corresponds to a cyclic query graph G^C. We will show that *every* spanning forest of J_Q^+ corresponds to a cyclic query graph, so that by Lemma 11, $Q \notin TQ$. Therefore $Q \in TQ$ must imply that every J^C corresponds to an acyclic query graph.

Consider a *simple* cycle in G^C, i.e., a cycle in which each node is incident with at most two edges in the cycle. The edges of this cycle map directly into edges of J^C; suppose this corresponding sequence of edges of J^C is $S = [(v_0, v_1), (v_2, v_3), \ldots, (v_{2n-2}, v_{2n-1})]$. Clearly, v_{2i-1} and v_{2i} are labeled by domains of the same relation for $i = 1, \ldots, n$ (arithmetic is mod $2n$). For each i, $1 \leq i \leq n$, if v_{2i-1} and v_{2i} are in the same connected component, then by the construction of canonical spanning forests, $v_{2i-1} = v_{2i}$. Thus S can be represented as a sequence of $p + 1$ paths, $p \geq 0$, $[[(v_{0,0}, v_{0,1}), (v_{0,1}, v_{0,2}), \ldots, (v_{0,m_0-1}, v_{0,m_0})], \ldots, [(v_{p,0}, v_{p,1}), \ldots, (v_{p,m_p-1}, v_{p,m_p})]]$, where each path is entirely contained in a connected component of J_Q^+.

We claim $p \geq 1$. Suppose $p = 0$. Since $v_{0,0}$ and v_{0,m_0} are in the same connected component of the join graph and are in the same relation, the construction of canonical spanning forests requires that $v_{0,0} = v_{m,0}$. But then the path in the canonical spanning forest is a cycle, a contradiction. So $p > 0$.

Consider one of the paths, $[(v_{i,0}, v_{i,1}), \ldots, (v_{i,m_i-1}, v_{i,m_i})]$. Since the transitive closure of any spanning forest of J_Q^+ equals the transitive closure of J^C, there must be a path from $v_{i,0}$ to v_{i,m_i} in all spanning forests of J_Q^+. This holds for $0 \leq i \leq p$. Hence for each spanning forest F there exists in F a sequence of $p + 1$ paths S_F with the same endpoints as the $p + 1$ paths of S.

To complete the proof, we must show that for each spanning forest F, the sequence S_F produces a cycle in its corresponding query graph. Since we chose a simple cycle in G^C, v_{m_i} and v_{m_j} are named by domains of different relations if $i \neq j$. Since $p > 0$, we have at least two distinct nodes in the query graph path that corresponds to S_F. So the path is a cycle. Since all spanning forests have a cycle, by Lemma 11 $Q \in TQ$, a contradiction. □

THEOREM 3. *Testing if $Q \in TQ$ can be decided in linear time.*

PROOF. The algorithm for testing if $Q \in TQ$ is

1. Construct a canonical spanning forest J^C of J_Q.
2. Construct the query graph G^C corresponding to J^C.
3. If G^C is acyclic, then answer $Q \in TQ$; else answer $Q \in CQ$.

By Lemma 12, the algorithm is correct. Both spanning forest and cycle detection can be computed in time linear in the number of edges of the graph [1]; so the algorithm has linear worst-case time. □

A tree query membership test when multiattribute semi-joins are permitted appears in [3].

We conclude by proving Lemma 8 from Section 4.3 as promised.

LEMMA 8. *Query Q with qualification q is in CQ iff q has a proper cyclic subqualification.*

PROOF. If. Let q' be a proper cyclic subqualification of q. When we construct the canonical spanning forest, in the third step of the **do**-loop we choose to include a clause of q' as an edge in a chain whenever possible. Clearly, every clause of q' can be added, since they are all in q^+. Also, by part (3) of the definition of proper cyclic subqualification, each connected component of the canonical spanning forest can contain at most one clause from q'; so, incorporating clauses from q' can never cause a cycle in a connected component. Since all clauses of q' are embedded in the spanning forest, the query graph corresponding to the forest has a cycle. By Lemma 11, $Q \in CQ$.

Only if. Suppose $Q \in CQ$. Construct a canonical spanning forest F for Q. By Lemma 11 there is a cycle in the corresponding query graph. Select such a cycle, and let C be the edges of F that correspond to the edges of the cycle. From C, construct a qualification q' as follows: For each path in C which is not a proper subpath of any other path in C whose endpoints are, say, $R_i.A$ and $R_j.B$, include $(R_i.A = R_j.B)$ in q'. Clearly q' is in q^+ and q' is proper cyclic. Also, if $(R_i.A = R_j.B)$ is in q^+ and $R_i.A$ and $R_j.B$ are in DOMAINS(q'), then $(R_i.A = R_j.B)$ is in q'; this follows because $R_i.A$ and $R_j.B$ must be in the same connected component of F, and each connected component of F has at most one clause in q' by construction, so q' is a proper cyclic subqualification of q as desired. □

6. *Conclusions*

Semi-joins are commonly used as basic operations in query processing algorithms, especially in a distributed environment. In these algorithms a strategy is simply a semi-join program. Our results show that for tree queries, as soon as a strategy embeds σ_Q, a full reduction and, hence, the full potential of semi-joins are achieved. Consequently the strategies tend to be short; at worst their length is bounded by a linear function of the number of relations referenced. Furthermore, tree-query membership can be tested in linear time. This suggests that searching for optimal strategies is quite likely to be easier for tree queries than for cyclic ones. We therefore recommend that the optimization problem for tree queries be treated as an important (and probably more tractable) special case of the general query processing problem.

For cyclic queries, finding good semi-join programs is likely to be quite difficult. Since cyclic queries are common too, we will either need to find new tactics (other

than semi-joins) for solving them or will probably have to be satisfied with heuristic approaches such as that of [14].

We emphasize, however, that our results only describe *worst-case* behavior of semi-join strategies that try to achieve *full* reductions. The pathological case that shows cyclic queries to behave badly may rarely occur. More important, full reductions are not always profitable, and a query optimizer should only produce the most profitable semi-join strategies. So while we have learned much about the strategy space of semi-join programs, the query optimization problem on "average" databases using the semi-join tactic remains open.

ACKNOWLEDGMENTS. We thank Wing S. Wong for simplifying our first version of the *TQ* membership algorithm, which resulted in the one presented in Section 5. Many notational improvements and clarifications over earlier drafts were contributed by Marco Casanova, Nat Goodman, and the referees. Finally, we thank M. Clarke, R. D'Arcangelo, and C. Louis for their expert preparation of this manuscript.

REFERENCES

1. AHO, A., HOPCROFT, J., AND ULLMAN, J. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass., 1974.
2. AHO, A., SAGIV, Y., AND ULLMAN, J.D. Equivalence among relational expressions. *SIAM J. Comput. 2* (May 1979), 218–246.
3. BERNSTEIN, P.A., AND GOODMAN, N. Full reducers for relational queries using multi-attribute semi-joins. Proc. Computer Networking Symposium, Gaithersburg, Md., 1979, pp. 206–215.
4. CHAMBERLIN, D.D., ASTRAHAN, M.M., ESWARAN, K.P., GRIFFITHS, P.P., LORIE, R.A., MEHL, J.W., REISNER, P., AND WADE, B.W. SEQUEL 2: A unified approach to data definition, manipulation, and control. *IBM J. Res. Dev. 20*, 6 (Nov. 1976), 560–575.
5. CHANDRA, A.K., AND MERLIN, P.M. Optimal implementation of conjunctive queries in relational data bases. Proc. 9th Ann. ACM Symp. on Theory of Computing, Boulder, Colo., May 1976, pp. 77–90.
6. CODD, E.F. A relational model of data for large shared data banks. *Commun. ACM 13*, 6 (June 1970), 377–387.
7. CODD, E.F. Relational completeness of data base sublanguages. In *Data Base Systems*, R. Rustin, Ed., Courant Computer Science Symposia Series, Vol. 6, Prentice-Hall, Englewood Cliffs, N.J., 1972, pp. 65–90.
8. HELD, G.D., STONEBRAKER, M., AND WONG, E. INGRES—A relational data base management system. *Proc. AFIPS 1975 National Computer Conf.*, AFIPS Press, Arlington, Va., 1975, pp. 409–416.
9. OZKARAHAN, E.A., SCHUSTER, S.A., AND SEVCIK, K.C. Performance evaluation of a relational associative processor. *ACM Trans. Database Syst. 2*, 2 (June 1977), 175–196.
10. ROTHNIE, J.B. Evaluating inter-entry retrieval expressions in a relational database management system. Proc. AFIPS 1975 National Computer Conf., AFIPS Press, Arlington, Va., 1975, pp. 417–423.
11. ROTHNIE, J.B., AND GOODMAN, N. A survey of research and development in distributed database management. *Proc. 3rd Inter. Conf. on Very Large Data Bases*, Tokyo, Japan, 1977, pp. 48–61.
12. SMITH, J.M., AND CHANG, P.Y.-T. Optimizing the performance of a relational algebra database interface. *Commun. ACM 18*, 10 (Oct. 1975), 568–579.
13. WONG, E., AND YOUSSEFI, K. Decomposition—A strategy for query processing. *ACM Trans. Database Syst. 1*, 3 (Sept. 1976), 223–241.
14. WONG, E. Retrieving dispersed data from SDD-1: A system for distributed databases. Proc. 1977 Berkeley Workshop on Distributed Data Management and Computer Networks, Berkeley, Calif., May 1977.

RECEIVED DECEMBER 1978; REVISED SEPTEMBER 1979; ACCEPTED SEPTEMBER 1979