COMMUTATIVITY AND ITS ROLE IN THE PROCESSING OF LINEAR RECURSION

Yannis E. Ioannidis †

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

Abstract

We investigate the role of commutativity in query processing of linear recursion. We give a sufficient condition for two linear, function-free, constant-free, and range-restricted rules to commute. The condition depends on the form of the rules themselves. For a restricted class of rules, we show that the condition is necessary and sufficient and can be tested in polynomial time in the size of the rules. Using the algebraic structure of such rules, we study the relationship of commutativity with several other properties of linear recursive rules. We show that commutativity is in the center of several important special classes of linear recursion, i.e., separable recursion and recursion with recursively redundant predicates.

1. INTRODUCTION

Several general algorithms have been proposed for the processing of recursive programs in database systems (DBMSs). Recursive query processing is recognized as an expensive operation, and all the proposed algorithms incur some significant cost. Thus, the importance of identifying special cases of recursion on which specialized and more efficient algorithms are applicable is obvious. Such special cases of recursion include bounded recursion (uniform and otherwise), transitive closure, separable recursion, and one-sided recursion. In this paper, we elaborate on another special case of recursion, where participating operators (or rules) commute with each other. When this happens, recursive queries can be decomposed into smaller queries, which are expected to have a lower total execution cost than the original query.

† Supported by the National Science Foundation under Grant IRI-8703592.

Proceedings of the Fifteenth International
Conference on Very Large Data Bases

Amsterdam, 1989

Commutativity has already been identified as a significant special case of recursion [Ioan88a]. It has been shown how several types of queries are affected by the applicability of commutativity and how the general algorithms for recursive queries are affected. Constants can also be used in conjunction with commutativity to reduce the amount of data the system has to look at to answer a query with selections. This earlier work on commutativity was done within the algebraic framework of linear recursive operators (rules) [Ioan86a, Ioan88a]. In this paper, we use the first order logic representation of rules to give necessary and sufficient conditions for two linear recursive rules of a restricted form to commute with each other. These conditions are based on the form of the rules themselves and make no direct use of the definition of commutativity, which requires composing the two rules in both ways and examining the two composites for equivalence. We also use the algebraic formulation of recursion to compare commutativity with other special classes of recursion, in particular, separable recursion and recursion with recursively redundant predicates, and discuss the effects of commutativity on the algorithms proposed for them.

The paper is organized as follows. Section 1 is an introduction. Section 2 is a summary of the algebraic model for linear recursion, which has been introduced elsewhere [Ioan86a, Ioan88a]. In Section 3, we define commutativity in the algebraic model and we compare the notion of commutativity with separability and recursive redundancy. Section 4 manipulates rules in the logic model and gives a decision algorithm for commutativity for a restricted class of rules. In Section 5, separability and recursive redundancy are reexamined for the restricted class of rules studied in Section 4. Finally, Section 6 presents our conclusions and gives some directions for future work.

2. ALGEBRAIC MODEL

In this section, we provide a summary of the algebraic model for linear recursion [Ioan86a, Ioan88a]. Consider a linear recursive rule

$$\mathbf{P}(\underline{x}^{(0)}) \wedge \mathbf{Q}_1(\underline{x}^{(1)}) \wedge \cdots \wedge \mathbf{Q}_k(\underline{x}^{(k)}) \rightarrow \mathbf{P}(\underline{x}^{(k+1)}), \quad (2.1)$$

where for each i, $\underline{x}^{(i)}$ is a vector of variables. No restriction is imposed on the form of the rule, or on the finiteness of the relations corresponding to the various predicates in the rule. Thus, for example, the rules can contain functions. Each one of $\mathbf{P}(\underline{x}^{(0)})$ and the $\mathbf{Q}_i(\underline{x}^{(i)})$'s is a (*positive*) *literal*. In relational terms, if $\{\mathbf{q}_i\}$ is a set of relations ($\mathbf{q}_i$

corresponding to the predicate Q_i), and $f(\mathbf{P}, \{\mathbf{q}_i\})$ is a function that accepts as input and produces as output relations of the same schema as $\mathbf{P}$, the above rule can be expressed as

$$f(\mathbf{P}, \{\mathbf{q}_i\}) \subseteq \mathbf{P},$$

or equivalently $\mathbf{P} \cup f(\mathbf{P}, \{\mathbf{q}_i\}) = \mathbf{P}$. Given the existence of an additional nonrecursive rule of the form

$$\mathbf{Q}(\underline{x}) \rightarrow \mathbf{P}(\underline{x}), \tag{2.2}$$

which in relational terms, given $\mathbf{q}$ a relation corresponding to $\mathbf{Q}$, is expressed as $\mathbf{q} \subseteq \mathbf{P}$, the problem of recursive inference can be stated as one of finding $\mathbf{P}$ such that

(a) $\mathbf{P} = f(\mathbf{P}, \{\mathbf{q}_i\}) \cup \mathbf{q}$,
(b) $\mathbf{P}$ is minimal with respect to (a), i.e., $\mathbf{P}'$ satisfying (a) implies $\mathbf{P} \subseteq \mathbf{P}'$.

The function $f(\mathbf{P}, \{\mathbf{q}_i\})$, having $\{\mathbf{q}_i\}$ as parameters and $\mathbf{P}$ as input, can be thought of as a linear relational operator applied to the recursive relation $\mathbf{P}$ to produce another relation of the same schema. Let R be the set of all such operators. We can establish an algebraic framework in which we can define operations on relational operators as follows. *Multiplication* of operators is defined by $(A * B)\mathbf{P} = A(B\mathbf{P})$ and *addition* by $(A+B)\mathbf{P} = A\mathbf{P} \cup B\mathbf{P}$. † For notational convenience we omit the operator $*$. The multiplicative identity ($1\mathbf{P} = \mathbf{P}$) and the additive identity ($0\mathbf{P} = \emptyset$, $\emptyset$ the empty set) are defined in obvious ways. The n-th power of an operator A is inductively defined as: $A^0 = 1$, $A^n = A * A^{n-1} = A^{n-1} * A$. Equality of operators in R is defined as $A = B \Leftrightarrow \forall \mathbf{P}, A\mathbf{P} = B\mathbf{P}$. Finally, a partial order $\leq$ is defined on R as $A \leq B \Leftrightarrow \forall \mathbf{P}, A\mathbf{P} \subseteq B\mathbf{P}$. The set R with the operations defined above forms a *closed semiring* [Ioan88a].

Having embedded the linear relational operators in the above algebraic framework of the closed semiring, the set of Horn clauses (2.1) and (2.2), assuming that A is the operator that corresponds to (2.1), can be rewritten as

$$A\mathbf{P} \subseteq \mathbf{P},$$

$$\mathbf{q} \subseteq \mathbf{P}.$$

The minimal solution of the system is the minimal solution of the equation

$$\mathbf{P} = A\mathbf{P} \cup \mathbf{q}. \tag{2.3}$$

† The above definitions are valid only if the operators involved are appropriately compatible, e.g., for +, the operators have to agree on the schema of their input and the schema of their output. Although in the rest of the paper we only deal with appropriately compatible operators, the general algebraic theory incorporates all operators [Ioan88a].

The solution is a function of $\mathbf{q}$. Hence, $\mathbf{P}$ can be written as $\mathbf{P} = B\mathbf{q}$, and the problem becomes one of finding the operator B. Manipulation of (2.3) results in the elimination of $\mathbf{q}$, so that the equation contains operators only. In this pure operator form, the recursion problem can be restated as follows. Given operator A, find B satisfying the following:

(a) $1 + AB = B$,
(b) B is minimal with respect to (a), i.e., $1 + AC = C \Rightarrow B \leq C$. (2.4)

Theorem 2.1: [Ioan88a] The solution of equation (2.4a) with restriction (2.4b) is $A^* = \sum_{k=0}^{\infty} A^k$.

The operator A^* is called the *transitive closure* of A. Theorem 2.1 is originally due to Tarski [Tars55], and in the database context, it was first examined by Aho and Ullman [Aho79b]. It is the first time though that the solution of (2.4) is expressed in an explicit algebraic form within an algebraic structure like the closed semiring of the linear relational operators. The implications of the manipulative power thus afforded on the implementation of A^* are significant [Ioan86a, Ioan86b, Ioan87, Ioan88a]. In this paper, we shall concentrate on the implications of commutativity of operators in the implementation of A^*.

Note that, although an operator A may be derived from a recursive rule, the operator itself is nonrecursive, i.e., it corresponds to a conjunctive query [Chan77]. Also note that A^* represents an operator. The query answer is the result of applying A^* to a given relation $\mathbf{q}$. This is only an abstraction, however, that allows us to study recursion within the closed semiring of relational operators. It poses no restriction whatsoever in the processing order of the query, i.e., it does not enforce that first A^* is computed and then it is applied to $\mathbf{q}$. For example, assume that A^* can be decomposed into B^* and C^*, i.e., $A^* = B^* C^*$, so that the final computation is $B^* C^* \mathbf{q}$. The computation may proceed by first computing C^*, then applying it to $\mathbf{q}$, and then applying *seminaive* [Banc85] with B as the basic operator and $(C^* \mathbf{q})$ as the initial relation. The significance of the algebraic formulation lies in the abstraction that it offers, within which the capability of the decomposition $A^* = B^* C^*$ can be exhibited.

3. COMMUTATIVITY VS. SEPARABILITY and RECURSIVE REDUNDANCY

3.1. Commutativity

We say that two operators B and C *commute* if $BC = CB$. Consider computing the transitive closure of A, A^*, where $A = B + C$. It has been shown that if $CB \leq B^k C^l$, for some k, l with $k \in \{0,1\}$ or $l \in \{0,1\}$, then $A^* = B^* C^*$ [Ioan88a]. Commutativity is a special case of this condition. The computation of A^* is decomposed into two smaller computations, those of B^* and C^* (plus an

additional multiplication of them). The complexity of B and C is smaller than that of A. In general, this is expected to affect the total cost of the computation significantly. Hence, it is important to be able to identify when two operators commute. In Section 4, we present a sufficient condition for commutativity, which for rules of some restricted form is shown to be necessary and sufficient.

3.2. Commutativity vs. Separability

Separable recursions have been identified by Naughton as an important class of linear recursion where efficient algorithms can be applied [Naug88]. In this section, we shall show that the efficient separable algorithm is applicable to the class of commutative recursions. For the sake of simplicity, we shall concentrate on two operators A_1 and A_2. The extensions of the results to an arbitrary number of operators is straightforward.

Theorem 3.1: Given two operators A_1 and A_2 that commute, and a selection σ that commutes with one of them, the separable algorithm can be used for the computation of $\sigma(A_1+A_2)^*$.

Proof: Let $A_1A_2=A_2A_1$. The transitive closure of the sum of A_1 and A_2 is given by $(A_1+A_2)^* = A_1^* A_2^*$ [Ioan88a]. Given an initial relation $\mathbf{q}$ and a query with a selection σ that commutes with A_1, we have

$$\sigma(A_1+A_2)^* \mathbf{q} = A_1^* (\sigma A_2^*)\mathbf{q}. \qquad (3.1)$$

To take advantage of the selection, the following algorithm may be used to derive the query answer given in (3.1). The variables B and C contain operators whereas the variables R and S contain relations. Multiplication of operators is shown explicitly for readability.

```
B := σ;
C := σ;
repeat
    B := B * A2;
    C := B + C;
until C doesn't change
R := C q;
S := R;
repeat
    R := A1 R;
    S := S ∪ R;
until S doesn't change
```

The first loop actually involves manipulating relations that are parameters of the various operators. If that is taken into account, and some small optimizations are incorporated so that, in every application of an operator inside each loop, only the new tuples produced in the previous iteration are used, the above algorithm is precisely the one proposed for separable recursions with full selections † [Naug88]. □

In general, given a set of operators $\{A_i\}$, $1 \le i \le n$, that are mutually commutative, and a set of selections $\{\sigma_i\}$, $0 \le i \le n$, such that σ_i commutes with all operators except A_i, the following holds:

$$\sigma_0\sigma_1\sigma_2 \cdots \sigma_n (A_1+A_2+\cdots+A_n)^* =$$

$$(\sigma_1 A_1)^* (\sigma_2 A_2)^* \cdots (\sigma_n A_n)^* \sigma_0.$$

Usually, most of the selections will not be present. In the presense of multiple selections, it is an interesting optimization problem to choose the order in which the various operators should be computed and the time when an operator should be applied to the input relation.

3.3. Commutativity vs. Recursive Redundancy

The class of recursions that contain recursively redundant predicates was also identified by Naughton [Naug86]. Consider the operator A that is the product of a set of operators $\{A_i\}$, i.e., $A=A_1A_2\cdots A_n$. In this case, every term in the series $A^* = \sum_{k=0}^{\infty} A^k$ is a product of the A_i's some number of times. An operator A_i, $0 \le i \le n$, is *recursively redundant* if there is some N such that each term in the series of A^* needs A_i factored in less than N times. The nonrecursive predicates appearing in A_i are also called recursively redundant. Before stating the main result of this subsection we need the following definitions. An operator B is *uniformly bounded*, if there exist K and N, $K<N$, such that $B^N \le B^K$. An operator B is *torsion*, if there exist K and N, $K<N$, such that $B^N = B^K$. Clearly, every torsion is uniformly bounded, but the opposite is not true in general. The effect of the presense of recursively redundant operators on the query processing algorithm of an operator is given by the following result [Ioan88a].

Theorem 3.2: If B and C are mutually commutative and B is torsion, then B is recursively redundant in $(B\ C)^*$.

Sketch of proof: Consider an operator A, such that $A = B\ C = C\ B$ and there exist K and N, $K<N$, such that $B^N = B^K$. Then,

$$A^* = (B\ C)^* = \sum_{m=0}^{K-1} B^m C^m + \left(\sum_{m=K}^{N-1} B^m C^m\right)(C^{N-K})^*.$$

The details of the above derivation are given elsewhere [Ioan88a]. Note that B^{N-1} is the highest power of B used in any term of A^*, i.e., B is recursively redundant. Clearly, the above formula corresponds to a more efficient algorithm than processing A as a whole, since B is processed only for a fixed finite number of times, i.e., $N-1$, beyond which only C is processed. □

† The precise definition of full selections is given by Naughton [Naug88]. The key observation is that if $A_1A_2=A_2A_1$ and $\sigma A_1=A_1\sigma$, then σ is a full selection.

4. CHARACTERIZATION OF COMMUTATIVITY

We now turn our attention to commutativity as expressed in a logic framework. We restrict ourselves to linear, function-free, constant-free, and *range-restricted* recursive rules, i.e., every variable in the consequent appears at least once in the antecedent as well. Thus, for any finite database, the answer to any query is finite. If a variable appears in the consequent of a rule, it is called *distinguished*, otherwise it is called *nondistinguished*. We assume that the rules have the same consequent and share no nondistinguished variables. Moreover, repeated variables in the consequent are replaced by distinct ones, while adding the appropriate equality predicates in the antecedent. Finally, although the original task is to compute the transitive closure of two recursive rules with the same consequent, we are interested in the commutativity of the underlying nonrecursive rules, i.e., conjunctive queries. (Commutativity as defined in Section 3, is a property of nonrecursive rules (operators).) Given a recursive rule, the corresponding underlying nonrecursive one will be written with $\mathbf{P_O}$ as the (output) predicate in its consequent and $\mathbf{P_I}$ as the (input) predicate in its antecedent. Nevertheless, we shall still be referring to these two predicates as instances of the recursive predicate.

We say that two rules r_1 and r_2 with the same consequent *commute* if composing r_1 with r_2, i.e., resolving the consequent of r_2 with the literal of the recursive predicate in the antecedent of r_1, (denoted by r_1r_2) and composing r_2 with r_1 (denoted by r_2r_1) give equivalent rules, i.e., given any relations for the predicates in their antecedents produce the same output relation for the predicate in their consequent. This, in turn, is equivalent to the existence of homomorphisms from each composite to the other [Chan77, Aho79a]. Given two rules r and s, a *homomorphism* $f: r \rightarrow s$ is a mapping from the variables of r into those of s, such that (i) if x is a distinguished variable then $f(x)=x$, and (ii) if $Q(x_1,...,x_n)$ appears in the antecedent of r, then $Q(f(x_1),...,f(x_n))$ appears in the antecedent of s. Clearly, the definition of commutativity suggests a straightforward algorithm to test it for two rules r_1 and r_2. The algorithm involves checking for equivalence of the two composites r_1r_2 and r_2r_1. Therefore, a polynomial time implementation of this algorithm is unlikely to exist, since equivalence of conjunctive queries is known to be an NP-complete problem [Chan77, Aho79a].

4.1. A Sufficient Condition

In this section, we shall give a sufficient condition for commutativity that avoids producing the two composites. The condition can be tested in exponential time, because it potentially involves testing for equivalence of conjunctive queries. The test, however, is still more efficient than the one based on the definition of commutativity, because the exponential part is only occasionally applied on parts of the original rules as opposed to always being applied on the composites of the two rules.

As a notation vehicle, we shall use a version of the α-graph of a rule (which we shall also call α-graph), which was introduced for the study of uniform boundedness [Ioan85]. The α-graph of a rule is defined as follows.

(i) For every variable in the rule, a node is put in the graph.

(ii) If two variables x, y appear in two consecutive argument positions of some nonrecursive predicate Q in the rule, an undirected edge $(x-y)$ is put in the graph between the corresponding two nodes x, y. Also, if x appears in a unary nonrecursive predicate Q in the rule, an undirected edge $(x-x)$ is put in the graph. In both cases, the label of the edge is Q.

(iii) If two variables x, y appear in the same position of the recursive predicate P in the antecedent and the consequent respectively, then a directed arc $(x \rightarrow y)$ is put in the graph from node x to node y.

The following definitions about the connected components of the (underlying undirected graph of the) α-graph of a rule are also necessary. A *persistent* component is one that contains exactly one variable, which is distinguished, and no nonrecursive predicates (undirected edges). Its variable is also called persistent. A *permutation* component is one that contains only distinguished variables (i.e., their positions in the antecedent is a permutation of their positions in the consequent) and no nonrecursive predicates (undirected edges). Its variables are also called permutation variables. Any other component is called a *general* component. Each one of its distinguished variables that is the head and the tail of a directed arc (i.e., it appears in the same position in the recursive predicate in the antecedent and the consequent) is called *semi-persistent*. Any other remaining distinguished variable is called general. Notice that every persistent component is a permutation component as well. Also, every persistent variable is both a semi-persistent and a permutation variable as well. Whenever it is necessary to exclude persistent components (variables) from the other classes, we shall qualify the terms by *nontrivial*, e.g., a nontrivial permutation component is a permutation component that is not persistent.

Example 4.1: The following is the α-graph of the rule

$$\mathbf{P}(u,v,w,x,y,z) :- \mathbf{P}(v,u,w,w,y,z) \wedge \mathbf{Q}(x,y).$$

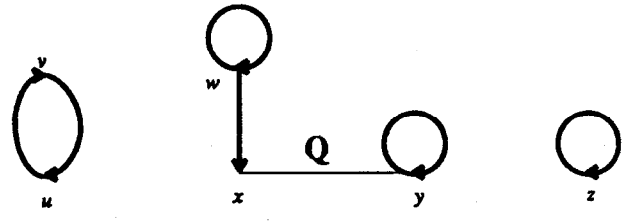

Figure 4.1: Example of an α-graph.

Variable z is persistent, variables w and y are semi-persistent, variables u and v are permutation, and variable x is general. □

For a rule r, we define the function h from the set of distinguished variables in r to the set of all variables in r. For a distinguished variable x, $h(x)$ is the variable that appears in the recursive predicate in the antecedent in the same position as x appears in the consequent. Since distinguished variables are assumed to appear exactly once in the consequents of rules (with the potential of repeated variables being realized by equalities in the antecedent), h is a function. Note that, if $h(x)=y$, then there exists a directed arc $(y \rightarrow x)$ in the α-graph of the rule. We also define powers of h as

$$h^1(x)=h(x), \text{ and}$$

$$h^n(x)=h(h^{n-1}(x)), \text{ if } h^{n-1}(x) \text{ is distinguished.}$$

For two rules r_1 and r_2, we define two more functions, g_{12} on the variables of r_2 and g_{21} on the variables of r_1. Since the two rules are assumed to share no nondistinguished variable, the former is defined as

$$g_{12}(z) = \begin{cases} z & z \text{ is nondistinguished} \\ h_1(z) & z \text{ is distinguished} \end{cases},$$

and similarly the latter. By definition, when $r_1 r_2$ is formed, a variable z in a predicate of r_2 is always replaced by $g_{12}(z)$.

Finally, two sets of permutation components are *consistent* if they are formed by the same (distinguished) variables, and for every variable x in them, it is $h_1(h_2(x)) = h_2(h_1(x))$. Clearly, consistent permutation components commute.

The following theorem gives a sufficient condition for commutativity of rules of the form specified in the beginning of Section 4. Its proof is omitted and can be found in the complete version of this paper [Ioan88b]. Another, less general sufficient condition for commutativity has been independently discovered and reported elsewhere [Rama89].

Theorem 4.1: Two rules r_1 and r_2 with the same consequent commute if

(a) the nontrivial semi-persistent variables in r_1 are semi-persistent in r_2 also (similarly for r_2),

(b) the nontrivial permutation variables in r_1 belong to consistent permutation components in r_2 (similarly for r_2),

(c) the general variables in r_1 either belong to an equivalent component in r_2, or they are persistent in r_2 (similarly for r_2).

Example 4.2: The following two rules, whose α-graphs are shown in Figure 4.2, commute with each other.

$$\mathbf{P_O}(x,y,z) :- \mathbf{P_I}(u,y,z) \wedge \mathbf{Q}(x,y)$$

$$\mathbf{P_O}(x,y,z) :- \mathbf{P_I}(x,y,v) \wedge \mathbf{R}(z,y)$$

Both composites are equal to the rule below.

$$\mathbf{P_O}(x,y,z) :- \mathbf{P_I}(u,y,v) \wedge \mathbf{Q}(z,y) \wedge \mathbf{R}(x,y)$$

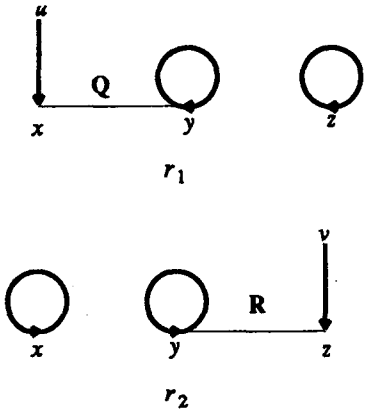

Figure 4.2: α-graphs of commuting rules satisfying the condition of Theorem 4.1.

Note that the condition of Theorem 4.1 is satisfied by the corresponding α-graphs. □

Unfortunately, as the following counter-example shows, the condition of Theorem 4.1 is not necessary for commutativity.

Example 4.3: The following two rules, whose α-graphs are shown in Figure 4.3, also commute with each other.

$$\mathbf{P_O}(x,y) :- \mathbf{P_I}(y,w) \wedge \mathbf{Q}(x)$$

$$\mathbf{P_O}(x,y) :- \mathbf{P_I}(u,v) \wedge \mathbf{Q}(x) \wedge \mathbf{Q}(y)$$

Both composites are isomorphic to the rule below.

$$\mathbf{P_O}(x,y) :- \mathbf{P_I}(u,v) \wedge \mathbf{Q}(y) \wedge \mathbf{Q}(w) \wedge \mathbf{Q}(x)$$

Figure 4.3: α-graphs of commuting rules not satisfying the condition of Theorem 4.1.

In this case, the condition of Theorem 4.1 is not satisfied. □

We are not aware of any necessary and sufficient condition for commutativity for rules of unrestricted form that is computationally or aesthetically better than the condition of the definition of commutativity. The following theorem shows that the condition of Theorem 4.1 is necessary and sufficient for commutativity if we restrict our attention to rules with *no repeated variables* in the consequent and *no repeated nonrecursive predicates* in the antecedent. The former restriction is enforced after all equalities have been eliminated from the antecedent. Before proceeding with the proof of the theorem, we need the following lemmas, whose proofs can be found in the complete version of this paper [Ioan88b].

Lemma 4.1: Consider two rules r_1 and r_2 with no repeated variables in the consequent and no repeated nonrecursive predicates in the antecedent that commute with each other. Let x be a distinguished variable, with $h_1(x)=x'$ and $h_2(x)=x''$, such that both x' and x'' are distinguished. Then, one of the following holds:

(a) both $h_1(x'')$ and $h_2(x')$ are distinguished and $h_1(x'')=h_2(x')$, i.e., $h_1(h_2(x))=h_2(h_1(x))$, or

(b) both $h_1(x'')$ and $h_2(x')$ are nondistinguished.

Lemma 4.2: Consider two rules r_1 and r_2 with no repeated variables in the consequent and no repeated nonrecursive predicates in the antecedent that commute with each other. Let $\{x_k\}$, $0\le k\le n+1$, be a set of distinguished variables such that $h_1(x_k)=x_{k+1}$, i.e., $h_1^{k+1}(x_0)=x_{k+1}$, for $0\le k\le n$, and x_0 appears in a nonrecursive predicate Q. Then, one of the following holds:

(a) $h_2(x_k)=x_k$, $0\le k\le n+1$, or

(b) $h_2(x_k)=x_{k+1}$, i.e., $h_2^{k+1}(x_0)=x_{k+1}$, for $0\le k\le n$, and x_0 appears in a nonrecursive predicate Q in r_2.

Theorem 4.2: Two rules r_1 and r_2 with the same consequent and no repeated variables in the consequent and no repeated nonrecursive predicates in the antecedent commute if and only if

(a) the nontrivial semi-persistent variables in r_1 are semi-persistent in r_2 also (similarly for r_2),

(b) the nontrivial permutation variables in r_1 belong to consistent permutation components in r_2 (similarly for r_2),

(c) the general variables in r_1 either belong to an equivalent component in r_2, or they are persistent in r_2 (similarly for r_2).

Proof: Recall that we assume that the two rules have the same consequents and share no nondistinguished variables. The "if" direction of the theorem follows from Theorem 4.1.

For the other direction of the theorem, assume that r_1 and r_2 commute. We shall show that for a distinguished variable x of r_1, one of (a), (b), or (c) holds in r_2, depending on the type of x. Since the theorem is symmetric in r_1 and r_2, the variables in r_2 are not examined. We shall always consider x being the first distinguished variable in the consequent, and we shall only be writing down the parts of the rules that are relevant to the proof. Also, unimportant variables will be denoted by _ .

(i) x is a nontrivial semi-persistent variable: In this case, x appears at least twice in the antecedent of r_1. This implies that it belongs to a general component, and that there exists a set of distinguished variables $\{x_k\}$, $0\le k\le n+1$, such that $h_1(x_k)=x_{k+1}$, $0\le k\le n$, with $x=x_n=x_{n+1}$, and x_0 appears in a nonrecursive predicate Q. If this is not true, then there must exist repeated variables in the consequent of r_1, which is a contradiction. Applying Lemma 4.2 for $x=x_n$ yields $h_2(x_n)=x_n$ or $h_2(x_n)=x_{n+1}$. Since $x=x_n=x_{n+1}$, this implies that in all cases $h_2(x)=x$, i.e., that x is semi-persistent in r_2.

(ii) x is a nontrivial permutation variable: If x is not a permutation variable in r_2 (i.e., it is a general variable or a nontrivial semi-persistent variable), then there exists a set of distinguished variables $\{y_k\}$, $0\le k\le n+1$, such that $h_2(y_k)=y_{k+1}$, $0\le k\le n$, with $x=y_{n+1}$, and y_0 appears in a nonrecursive predicate Q in r_2. By Lemma 4.2, this implies that either $x=h_1(x)$ or $x=h_1^{n+1}(y_0)$ and y_0 appears in a nonrecursive predicate Q in r_1. In the first case, x is a semi-persistent variable in r_1, and in the second case, x is a general variable in r_1. In both cases, this is a contradiction, since x is a nontrivial permutation variable in r_1. Hence, x must be a permutation variable in r_2 also.

Since x is a permutation variable in r_1 and r_2, $h_1(x)$ and $h_2(x)$ must also be permutation variables in r_1 and r_2 respectively ($h_i(x)$ is part of the same permutation as x in r_i). The argument in the previous paragraph can be applied in the case of $h_2(x)$ also and yield that $h_2(x)$ is a permutation variable in r_1 as well. Hence, $h_1(x)$, $h_2(x)$, and $h_1(h_2(x))$ are distinguished variables. By Lemma 4.1, $h_2(h_1(x))$ is also distinguished, and $h_2(h_1(x))=h_1(h_2(x))$. This implies that x belongs to consistent sets of permutation components in r_1 and r_2.

(iii) x is a general variable: This implies that there exists a set of distinguished variables $\{x_k\}$, $0\le k\le n+1$, such that $h_1(x_k)=x_{k+1}$, $0\le k\le n$, with $x=x_{n+1}$, and x_0 appears in a nonrecursive predicate Q. By Lemma 4.2, this implies that either $x=h_2(x)$, i.e., that x is a semi-persistent variable in r_2, or $x=h_2^{n+1}(x_0)$, and x_0 appears in a nonrecursive predicate Q in r_2, i.e., that x is a general variable in r_2. We shall examine the two cases separately.

If x is semi-persistent in r_2, we shall show that it cannot be nontrivial semi-persistent, i.e., it must be persistent. Assume to the contrary that x is nontrivial semi-persistent in r_2. From case *(i)* for r_2, we conclude that x is semi-persistent in r_1, which is a contradiction. Hence, x must be persistent.

If x is general in r_2, we shall show that it belongs to a component that is equivalent to its component in r_1. Recall

that we examine the case where $h_2(x_k)=x_{k+1}$, for all $0 \le k \le n$, which implies that $h_1(x_k)=h_2(x_k)$. Since $x=x_{n+1}$ is an arbitrary variable in its component, we may conclude that for any distinguished variable z in that component, either both $h_1(z)$, $h_2(z)$ are distinguished and $h_1(z)=h_2(z)$, or both $h_1(z)$, $h_2(z)$ are nondistinguished, i.e., the structure of h for the components of z in r_1 and r_2 is the same. Hence, if we assume that the two components are not equivalent, there must be some nonrecursive predicate connected (through a series of nonrecursive predicates) to a distinguished variable in the component in r_1 that is not connected through the same series of nonrecursive predicates to the same distinguished variable in the component in r_2 (or vice-versa). Without loss of generality, assume that x is such a distinguished variable. Also without loss of generality, assume that $h_1(x)=h_2(x)=y$ is a distinguished variable, and that only nondistinguished variables appear in the nonrecursive predicates connected to x (except x). The other cases are treated similarly. This situation is depicted by the rules of Figure 4.4 and their composites. Clearly, since $y \neq x$ (x is general and not semi-persistent), the two composites are not equivalent, and r_1 and r_2 cannot commute, which is a contradiction. Hence, the assumption that the two components where x belongs in r_1 and r_2 are not equivalent is wrong. □

The complexity of the condition in Theorem 4.2 is given by the following theorem.

Theorem 4.3: Commutativity of two rules with no repeated variables in the consequent and no repeated variables in the antecedent can be tested in $O(a_r + n\,a_n + n^2)$ time, where a_r is the arity of the recursive predicate, a_n is the maximum arity of a nonrecursive predicate, and n is the maximum number of the nonrecursive predicates in the rules.

Sketch of proof: The algorithm has the following basic steps.

(a) Identify the connected components of the underlying undirected graphs of the α-graphs of the two rules, while identifying the type of every distinguished variable (i.e., persistent, nontrivial semi-persistent, nontrivial permutation, or general). The quantity $a_r + n\,a_n$ is an upper bound on both the nodes and the edges/arcs in the graph. So, this step can be done in $O(a_r + n\,a_n)$ time [Aho74].

(b) For every nontrivial semi-persistent variable in the one rule, check if it is semi-persistent in the other. This step takes $O(1)$ for every nontrivial semi-persistent variable.

(c) For every nontrivial permutation variable in the one rule, check if it belongs in a consistent permutation component in the other. This step takes $O(1)$ for every nontrivial permutation variable.

(d) For every general variable in the one rule, check if it is persistent in the other. If it is, do nothing. This step takes $O(1)$ for every such variable. If it is not, check if it belongs in an equivalent component in the other rule. Because the rules contain no repeated nonrecursive predicates in the antecedent, equivalence can be tested in polynomial time as follows (the components have to be isomorphic).

(d1) For every distinguished variable, check if the h functions of the two rules are compatible. This step takes $O(a_r)$ time.

(d2) For every nonrecursive predicate in the one rule, check if it exists in the other rule and if the mappings between the variables are consistent with the mappings of the other predicates. This step takes $O(n)$ time for the first check and $O(a_n)$ time for the second check for every nonrecursive predicate, for a total of $O(n\,a_n + n^2)$ time.

---

r_1: $\mathbf{P_O}(x,\cdots) :- \mathbf{P_I}(y,\cdots) \wedge \mathbf{R}_1(x,z_1) \wedge \mathbf{R}_2(z_1,z_2) \cdots \wedge \mathbf{R}_{m-1}(z_{m-2},z_{m-1}) \wedge \mathbf{R}_m(z_{m-1},z_m) \wedge \cdots$

r_2: $\mathbf{P_O}(x,\cdots) :- \mathbf{P_I}(y,\cdots) \wedge \mathbf{R}_1(x,z'_1) \wedge \mathbf{R}_2(z'_1,z'_2) \cdots \wedge \mathbf{R}_{m-1}(z'_{m-2},z'_{m-1}) \wedge \cdots$

r_1r_2: $\mathbf{P_O}(x,\cdots) :- \mathbf{P_I}(\_,\cdots) \wedge \mathbf{R}_1(y,z'_1) \wedge \mathbf{R}_2(z'_1,z'_2) \cdots \wedge \mathbf{R}_{m-1}(z'_{m-2},z'_{m-1}) \wedge$
$\mathbf{R}_1(x,z_1) \wedge \mathbf{R}_2(z_1,z_2) \cdots \wedge \mathbf{R}_{m-1}(z_{m-2},z_{m-1}) \wedge \mathbf{R}_m(z_{m-1},z_m) \wedge \cdots$

r_2r_1: $\mathbf{P_O}(x,\cdots) :- \mathbf{P_I}(\_,\cdots) \wedge \mathbf{R}_1(y,z_1) \wedge \mathbf{R}_2(z_1,z_2) \cdots \wedge \mathbf{R}_{m-1}(z_{m-2},z_{m-1}) \wedge \mathbf{R}_m(z_{m-1},z_m) \wedge$
$\mathbf{R}_1(x,z'_1) \wedge \mathbf{R}_2(z'_1,z'_2) \cdots \wedge \mathbf{R}_{m-1}(z'_{m-2},z'_{m-1}) \wedge \cdots$

Figure 4.4: Rules with a general variable in nonequivalent components and their composites.

The total time for steps (b), (c), and (d) without (d1) and (d2) is $O(a_r)$. If we add to that the time needed for (a), (d1), and (d2), we conclude that the total running time of the test is $O(a_r + n\, a_n + n^2)$. □

5. SEPARABILITY AND RECURSIVE REDUNDANCY REVISITED

In Section 3, we examined commutativity vs. separability and recursive redundancy as expressed in the abstract form of the algebra to obtain results that hold for any linear rules. In this section, we restrict ourselves to function-free, constant-free, and range-restricted rules and use our results from Section 4 to obtain more relationships of commutativity with separability and recursive redundancy for this class of rules.

5.1. Commutativity vs. Separability

Two rules r_1 and r_2 with the same consequent are separable [Naug88] if

(1) for any distinguished variable x, either $h_i(x)=x$ or $h_i(x)$ is nondistinguished, $i=1,2$,

(2) for any distinguished variable x, either both x and $h_i(x)$ appear under nonrecursiver predicates or none, $i=1,2$, and

(3) the sets of distinguished variables that appear under nonrecursive predicates in r_1 and r_2 are either equal or have an empty intersection.

We ignore a fourth clause that the original definition contained, since it was identified as nonessential for the correctness of the separable algorithm. For the case of two rules, one can take advantage of the efficient features of the separable algorithm only if in clause (3) the intersection of the sets of distinguished variables that appear under nonrecursive predicates in r_1 and r_2 is empty. With this assumption, we can prove the following lemma.

Lemma 5.1: If two rules r_1 and r_2 with the same consequent are separable, then they only contain semi-persistent and general variables, and any general or nontrivial semi-persistent variable in r_1 is persistent in r_2 (similarly for the variables of r_2).

Combining Lemma 5.1 with Theorem 4.1 yields the following theorem.

Theorem 5.1: If two rules are separable then they commute, but the opposite does not hold.

Proof: If two rules r_1 and r_2 are separable, by Lemma 5.1, the general variables of r_1 satisfy condition (c) of Theorem 4.1, and the nontrivial semi-persistent variables of r_1 satisfy condition (a) of the same theorem (similarly for the variables of r_2). Moreover, by the same lemma, there are no other types of variables in r_1 or r_2. Thus, by Theorem 4.1, the two rules commute.

The rules of Example 4.2 serve as examples of commutative rules that are not separable. They violate both conditions (2) and (3) of the separable definition. □

By Theorem 5.1, commutativity is a strictly more general notion than separability. Nevertheless, all the efficient processing algorithms for separable recursions are applicable for commutative rules as well (Theorem 3.1).

5.2. Commutativity vs. Recursive Redundancy

Given the α-graph of a rule, the *augmented* α-graph is produced by disconnecting all edges emanating from semi-persistent variables in the α-graph, and replacing the semi-persistent variable with a distinct nondistinguished variable for every such edge [Naug86]. A necessary and sufficient condition for a nonrecursive predicate in a rule of some restricted form to be redundant is given by the following theorem.

Theorem 5.2: [Naug86] A nonrecursive predicate in a rule with no repeated variables in the consequent and no repeated nonrecursive predicates in the antecedent is recursively redundant if and only if it appears in a bounded component of the augmented α-graph of the rule.

Theorems 3.2, 4.2, and 5.2 imply the following.

Theorem 5.3: Let $A=B\,C$ be an operator corresponding to a rule with no repeated variables in the consequent and no repeated nonrecursive predicates in the antecedent. Then, B is recursively redundant if and only if B and C commute and B is uniformly bounded.

Sketch of proof: From Theorem 4.2, we can prove that each component in the augmented α-graph, seen as an operator, commutes with the operators of all the other components. We can also show that, if the rules contain no repeated variables in the consequent and no repeated nonrecursive predicates in the antecedent, any uniformly bounded component of the augmented graph is torsion. Combining this with Theorem 5.2, yields that the condition of Theorem 3.2 is necessary and sufficient (in this case the properties of being torsion and uniformly bounded coincide). □

6. CONCLUSIONS

We have investigated the role of commutativity in query processing of linear recursive rules. Using the algebraic structure of such rules, we have identified commutativity as the essence of many properties that give rise to important classes of recursive rules, i.e., separable rules and rules with recursively redundant predicates. Focusing on range-restricted rules that contain no functions, and no constants, we have given a sufficient condition for such rules to commute. We have also shown that the condition is necessary and sufficient when the rules contain no repeated variables in the consequent and no repeated nonrecursive predicates in the antecedent. In that case, the condition can be tested in polynomial time in the size of the rules.

Commutativity emerges as a key property of linear recursive rules for which efficient algorithms can be applied. This paper is a first step in the investigation of its power. We believe that there is much more work to be done in this direction. Some problems we plan to study in the future are the following: characterize commutativity in more general classes of rules than the one studied in this paper; investigate the relationship of commutativity and one-sided recursion; investigate the relationship of commutativity and several optimizations proposed for the magic sets and counting algorithms (e.g., there seems to be a strong relationship between commutativity and the semijoin optimization [Beer87]); examine ways to take advantage of partial commutativity, i.e., when the transitive closure of a product of operators is to be computed, only a subset of which are mutually commutative; and examine ways to take advantage of commutativity appearing in some higher power of an operator.

7. REFERENCES

[Aho74]
Aho, A., J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison Wesley, Reading, MA, 1974.

[Aho79a]
Aho, A., Y. Sagiv, and J. Ullman, "Equivalences Among Relational Expressions", *SIAM Computing Journal* **8**, 2 (May 1979), pp. 218-246.

[Aho79b]
Aho, A. and J. Ullman, "Universality of Data Retrieval Languages", in *Proc. of the 6th ACM Symposium on Principles of Programming Languages*, San Antonio, TX, January 1979, pp. 110-117.

[Banc85]
Bancilhon, F., "Naive Evaluation of Recursively Defined Relations", in *Proc. of the Islamorada Workshop on Large Scale Knowledge Base and Reasoning Systems*, Islamorada, FL, February 1985.

[Beer87]
Beeri, C. and R. Ramakrishnan, "On the Power of Magic", in *Proc. of the 6th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems*, San Diego, CA, March 1987, pp. 269-283.

[Chan77]
Chandra, A. K. and P. M. Merlin, "Optimal Implementation of Conjunctive Queries in Relational Data Bases", in *Proc. 9th Annual ACM Symposium on Theory of Computing*, Boulder, CO, May 1977, pp. 77-90.

[Ioan85]
Ioannidis, Y. E., "A Time Bound on the Materialization of Some Recursively Defined Views", in *Proc. 11th International VLDB Conference*, Stockholm, Sweden, August 1985, pp. 219-226.

[Ioan86a]
Ioannidis, Y. E. and E. Wong, "An Algebraic Approach to Recursive Inference", in *Proc. of the 1st International Conference on Expert Database Systems*, Charleston, South Carolina, April 1986, pp. 209-223.

[Ioan86b]
Ioannidis, Y. E., "On the Computation of the Transitive Closure of Relational Operators", in *Proc. 12th International VLDB Conference*, Kyoto, Japan, August 1986, pp. 403-411.

[Ioan87]
Ioannidis, Y. E. and E. Wong, "Query Optimization by Simulated Annealing", in *Proc. of the 1987 ACM-SIGMOD Conference on the Management of Data*, San Francisco, CA, May 1987, pp. 9-22.

[Ioan88a]
Ioannidis, Y. E. and E. Wong, *"Towards an Algebraic Theory of Recursion"*, Technical Report No. 801, University of Wisconsin, Madison (submitted for publication), October 1988.

[Ioan88b]
Ioannidis, Y. E., *"Commutativity and its Role in the Processing of Linear Recursion"*, Technical Report No. 804, University of Wisconsin, Madison, November 1988.

[Naug86]
Naughton, J., "Redundancy in Function-Free Recursive Rules", in *Proc. of the 3rd Symposium on Logic Programming* , September 1986, pp. 236-245.

[Naug88]
Naughton, J., "Compiling Separable Recursions", in *Proc. of the 1988 ACM-SIGMOD Conference on the Management of Data*, Chicago, IL, June 1988, pp. 312-319.

[Rama89]
Ramakrishnan, R., Y. Sagiv, J. D. Ullman, and M. Vardi, *"Proof Tree Transformation Theorems and their Applications"*, Philadelfia, PA, Proc. of the 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 1989, pp. 172-181.

[Tars55]
Tarski, A., "A Lattice Theoretical Fixpoint Theorem and its Applications", *Pacific Journal of Mathematics* **5** (1955), pp. 285-309.