Data Partition and Parallel Evaluation of Datalog Programs

Weining Zhang, Ke Wang, and Siu-Cheung Chau

Abstract — **Parallel bottom-up evaluation provides an alternative for the efficient evaluation of logic programs. Existing parallel evaluation strategies are neither effective nor efficient in determining the data to be transmitted among processors. In this paper, we propose a different strategy, for general Datalog programs, that is based on the partitioning of data rather than that of rule instantiations. The partition and processing schemes defined in this paper are more general than those in existing strategies. A parallel evaluation algorithm is given based on the semi-naive bottom-up evaluation. A notion of potential usefulness is recognized as a data transmission criterion to reduce, both effectively and efficiently, the amount of data transmitted. Heuristics and algorithms are proposed for designing the partition and processing schemes for a given program. Results from an experiment show that the strategy proposed in this paper has many promising features.**

Index Terms — **Database, logic programs, parallel processing, data partition, data transmission criteria, algorithm.**

I. Introduction

Query optimization [1], [9] has been the only approach studied for efficient bottom-up evaluation of Datalog programs until recently when parallel evaluation strategies [12], [14], [3], [5], [6], [13], [8] were proposed as alternative solutions to the problem. These strategies are based on *program restriction* [12]. The idea is to view the workload of the bottom-up program evaluation as a set of rule instantiations and to partition these rule instantiations among a set of processors by appending conditions to rule bodies. These strategies differ from each other in the amount of data transmitted. Strategies in [12], [14], [3], [5], [8], are for *decomposable programs* and require no data transmission. However, decomposable programs form rather restricted classes and the decomposability of general Datalog programs is undecidable. Strategies in [13], [6], on the other hand, are for general Datalog programs and require data transmission. Based on these general strategies, each processor evaluates a set of restricted rules obtained from the given program using data in the local database. At each iteration of the parallel semi-naive algorithm of the strategies, the data that are newly derived at a processor and satisfy a data transmission criterion are sent to other processors for further evaluation. The evaluation terminates when no processor is busy and no data is in transmission.

A good data transmission criterion for such strategies should preserve a correct computation, allow a minimal amount of data to be transmitted, and be effective and efficient in testing data. However, the data transmission criteria of existing strategies do not meet all these requirements. For example in [13], a tuple is sent from processor p_i to processor p_j only if it is computed at p_i and belongs to a set SR_j whose definition is quoted below.

> An S-fact, f, is in SR_j if and only if: (Condition K1) there is some rule of the program P, say r_g, such that f is not in r_g, but there is an instantiation of it that satisfies the predicate h_{gj}, and f appears in the body of the instantiated rule. In other words, a tuple f is in SR_j, if there is an instantiation for which p_j is in charge, that uses f.

Here h_{gj} is the restricting predicate of rule r_g for processor p_j. Condition K1 may have the following problems. If h_{gj} contains more variables than f can instantiate, it becomes necessary to determine if there are valid values to instantiate the extra variables. By valid values we mean those values that are in the domain of the variables and make h_{gj} true. If the rule instantiation is allowed to be arbitrary, that is, it does not require the instantiated subgoals to be true wrt the given database, in the worst case, a search in domains of those variables may still be needed to find valid values. Furthermore, a lot of irrelevant data will be sent. To see this, consider the following rule with h_{gj} being $h(x, y, z) = j$, where h is some type of function.

$$q(x, y) :- q(x, w), b(w, w), c(y, z, w).$$

To see if a tuple $q(s, t)$ satisfies Condition K1, the following instantiation of variables is needed: $x = s$, $w = t$, $y = m$, and $z = n$, for some values m and n. Obviously, m and n must be in the domains of y and z, respectively, and must be chosen so that $h(s, m, n) = j$. In a worst case where, for example, both y and z have some limited, finite domains, and the equation $h(x, y, z) = j$ does not have a unique solution of y and z for given x and j, some kind of search for suitable values m and n in their respective domains is needed. If the instantiation can be found, $q(s, t)$ will be sent even though $b(t, t)$ or $c(m, n, t)$ may not be in the database at all. However, if the rule instantiation is required to be successful, that is, every instantiated

Manuscript received January 2, 1993. Manuscript revised June 12, 1993. This work was supported in part by the Natural Sciences and Engineering Research Council of Canada.

Weining Zhang and Siu-Cheun Chau are with the Department of Mathematics and Computer Science, University of Lethbridge, Lethbridge, Alberta, Canada, T1K 3M4. Dr. Zhang can be reached vie e-mail at zhang@cs.uleth.ca. Ke Wang is with the Department of Information Systems and Computer Science, National University of Singapore, 10 Kent Ridge Crescent, Singapore, 0511.

IEEE Log Nunber K95013

subgoal must be true wrt the given database, although less irrelevant data will be sent, Condition K1 may cause several difficulties. One difficulty is that in the worst case the time needed to find valid values becomes proportional to the cross product of a number of relations. A second difficulty is that if some of the extra variables are in recursive subgoals or in subgoals whose relations are distributed over the network, it can not be sure that a tuple f will not be needed at p_j even if no rule instantiation containing f and satisfying h_{gj} is found when Condition K1 is tested. In [6], the strategy is obtained through a program rewriting. A set of sending rules is constructed to define the data to be transmitted. For an IDB predicate C, the set of tuples of C sent from processor i to processor j is defined by sending rules of the form: $C_{ij}(\vec{y}) :- C^i_{out}(\vec{y}), h(v(r)) = j$, where C^i_{out} is the set of tuples of C derived at processor i, $C(\vec{y})$ is a subgoal in some rule r, and $h(v(r))$ is a discriminating function on a sequence of variables $v(r)$ in r. In order to prevent processor i from performing the computation that is performed later at processor j, all variables in $v(r)$ must appear in at least one subgoal of rule r [6]. Since $C(\vec{y}$ may not be a subgoal containing all variables in $v(r)$, sending rules may not be safe. Even if it is safe, it is still not clear how to evaluate a sending rule without considering all instantiations of the variables that are in $v(r)$ but not in $\vec{y}$.

In this paper, we propose a parallel evaluation strategy for general Datalog programs based on partitioning of data. We present partition and processing schemes that allow multiple partition functions to be specified over each rule. A notion of potential usefulness is used as a data transmission criterion. We show that although not effective and efficient for general partition and processing schemes, the potential usefulness can be used to test a tuple in a time that is independent of the size of the database, if all partition functions are local (see Definition 4.7 and Proposition 4.1). A parallel algorithm based on the semi–naive bottom–up evaluation is given to evaluate restricted programs of a processing scheme. Issues of designing evaluation strategies are studied. Heuristics and algorithms for the design process are given. Results from an experiment of performance study is also provided. Our strategy is more general and more practical than previous strategies.

To simplify the problem, our approach assumes an all–to–all communication network or a multiprocessor environment. Such an assumption is common in other related works such as [2], [3], [5], [8], [13]. The implementation of our and other known parallel evaluation strategies on other network topologies is itself an interesting and independent problem. A comparison of some sequential and parallel processing strategies for the single source reachability problem of directed graphs (defined by database relations) is recently presented in [11]. More work needs yet to be done in this area.

The rest of the paper is organized as follows. In Section 2, we introduce basic concepts and the notation used throughout the paper. In Section 3, we define partition and processing schemes. The potential usefulness, as a data transmission criterion is presented in Section 4. A parallel evaluation algorithm is given in Section 5. In Section 6, we consider issues of designing partition and processing schemes for a given program. In Section 7, we present results from an experiment that is intended to study the workload distribution, communication costs, and speedup factors of different strategies. Section 8 concludes the paper.

II. PRELIMINARY

The conventional terminology is used in this paper. The reader may refer to Ullman [9] for more complete treatment of the basic concepts and terminology.

An *atom* is of the form $q(\vec{x})$, where q is a predicate name and $\vec{x}$ is a vector of variables and constants. An atom with no variable is called a *tuple*. A *constraint* is of one of the following forms: $x\ \sigma\ y$, $x\ \sigma\ c$, or $f(\vec{z})\ \sigma\ c$, where x and y are variables, c is a constant, $\sigma \in \{=, \neq, >, \geq, <, \leq\}$ is a built–in predicate, $\vec{z}$ is a vector of variables, and $f(\vec{z})$ is a function that maps the domains of variables in $\vec{z}$ to a subset of values in the database domain on which the relationship σ is defined. For example, $f(x, y) \stackrel{\text{def}}{=} (x + y) \bmod 5$, where the domains of x and y are integers, is a function. Each function is associated with one (predefined) mapping, thus it differs from a function symbol in the conventional logic. A *rule* is of the form $a :- b_1, b_2, \ldots, b_k$, where the *head* a is an atom, and each *subgoal* b_i is either an atom or a constraint. If b_i is an atom, it is also called an *ordinary subgoal*. The conjunction of bs is called the *body* of the rule. A rule is *safe* if every variable in the rule is limited [9], that is, in an ordinary subgoal or equated to a constant or a limited variable through a chain of equalities (that is, =). A *program* is a finite set of safe rules. A *Datalog program* is a program without function. Those predicates that appear only in rule bodies of a program are *extentional database* (*EDB*) predicates, and those that appear in the head of some rule are *intentional database* (IDB) predicates. An IDB (respectively, EDB) subgoal (respectively, tuple and relation) is a subgoal (respectively, tuple and relation) associated with an IDB (respectively, EDB) predicate. An EDB is simply a finite set of EDB tuples. A rule without IDB subgoal is an *exit* rule.

A *(ground) substitution* is a set $\theta = \{v_1/c_1, \ldots, v_n/c_n\}$, where $\{v_1, \ldots, v_n\}$ is a set of variables and $c_1, \ldots, c_n$ are constants. Let E be a rule (respectively, an atom, a constraint, and a function) and $\theta = \{v_1/c_1, \ldots, v_n/c_n\}$ be a substitution. The *instantiation* of E by θ, denoted by $E\theta$, is the rule (respectively, the atom, the constraint, and the function) obtained from E by simultaneously replacing each occurrence of v_i in E by c_i. Given an EDB, a tuple t is *derivable* using a program $\mathcal{P}$ if

1) t is in the EDB, or
2) there is a rule r and a substitution θ such that the head of $r\theta$ is t, each ordinary subgoal of $r\theta$ is a tuple derivable using $\mathcal{P}$, and each constraint in $r\theta$ is true under the conventional interpretation. We call such a rule instantiation a *successful (rule) instantiation* of r.

The set of all tuples derivable using a program $\mathcal{P}$ is the *least fixpoint* of $\mathcal{P}$ [7], [9], denoted by *lfp*($\mathcal{P}$), and can be computed using the semi–naive bottom–up evaluation [1], [9].

To simplify the presentation, we consider rectified rules.

Rules for a predicate p are *rectified* if their heads are of the identical form $p(x_1, \ldots, x_k)$ with distinct variables $x_1, \ldots, x_k$, and for each rule, every argument of each ordinary subgoal is a variable appearing exactly once in the ordinary subgoal in the rule. A program is rectified if every rule in the program is rectified. This definition differs from the one in [9] since it requires not only the head but also all ordinary subgoals of each rule to contain neither constant nor repeated variable. Rules can be rectified by first applying the rectification of [9] and then for each rule, replacing each occurrence of a repeated variable or a constant, in an ordinary subgoal, with a distinct new variable and appending to the rule body an equality $x = y$ if an occurrence of x is replaced by y. It is possible to minimize the number of new variables needed and to complete the process by scanning the rule body exactly once from left to right. Since the detail is not important to this paper, it is omitted. The following is an example of a rule and its rectified counterpart:

$$p(x, y, y) :- e(x, 4, u, v), p(u, w, w), p(y, v, m), f(m, v, u).$$
$$p(z_1, z_2, z_3) :- e(z_1, t, u, v), p(u_1, w, w_1), p(z_2, v_1, m), f(m_1, v_2, u_2), z_2 = z_3, t = 4, u = u_1, v = v_1, u_1 = u_2, w = w_1, v_1 = v_2, m = m_1.$$

III. Partition and Processing Schemes

In this and the next two sections, we present a data partition paradigm of parallel evaluation of Datalog programs. Given a rectified Datalog program, a set of restricted programs is constructed to partition the computation workload.[1] Each restricted program represents a fraction of the computation and is evaluated at exactly one processor. During the evaluation, data are transmitted, according to a data transmission criterion, among processors through either a communication network or a shared memory. There are three important issues, namely, partition and processing schemes, data transmission criteria, and parallel evaluation algorithms. This section deals with the first issue. The following example provides a motivation. For convenience, we assume a database domain of integers.

Example 3.1: Consider the following (not–yet–rectified) Datalog program.

1*a*: q(x, y, z) :- q(u, x, v), q(y, v, w), q(w, z, t).
1*b*: q(x, y, z) :- e(x, y, z).

Restricted programs can be created by defining one function for each rule. A possible function for rule 1*a* is $(u + x + v) \bmod n$, where n is the number of processors defining the range of the function. Restricted versions of rule 1*a* are of the form

2*a*: $q(x, y, z) :- q(u, x, v), q(y, v, w), q(w, z, t), (u + x + v) \bmod n = i.$

[1] In this paper, we consider only Datalog programs and their restricted programs although the results presented in this paper also apply to programs with conventional function symbols.

where $0 \le i \le n-1$. Each i corresponds to a restricted version of 1*a*. Suppose Q is the relation associated with predicate q. The evaluation of rules 1*a* and 2*a* can be considered as a repeated search for successful rule instantiations in which each ordinary subgoal is a tuple in Q. Since the set of successful instantiations of 2*a* must satisfy the additional constraint, it is a subset of that of 1*a*. For rule 1*a*, the search space is represented by the cross product $Q \times Q \times Q$. For rule 2*a*, $(u + x + v) \bmod n = i$ induces a partition on relation Q such that for any i, a tuple $q(a,b,c)$ in Q is in a fragment $F_{<i>}$ iff $(a + b + c) \bmod n = i$. Since only tuples in $F_{<i>}$ can possibly be the instantiation of $q(u, x, v)$ in any successful instantiation of 2*a*, the search space is reduced to $F_{<i>} \times Q \times Q$.

Another way of restricting 1*a* is to use two functions and have restricted rules of the following form:

3*a*: $q(x,y,z) :- q(u, x, v), q(y, v, w), q(w, z, t), v^2 \bmod n_1 = i, w \bmod n_2 = j.$

where n_1 and n_2 are integers. The two constraints induce three different partitions of Q each of which corresponds to an ordinary subgoal. Namely, for any i and j, let the fragments corresponding to $q(u, x, v)$, $q(y, v, w)$ and $q(w, z, t)$ be $F_{<i>}$, $T_{<i,j>}$ and $S_{<j>}$, respectively. A tuple $q(a, b, c)$ in Q is in $F_{<i>}$ iff $c^2 \bmod n_1 = i$; it is in $T_{<i,j>}$ iff $b^2 \bmod n_1 = i$ and $c \bmod n_2 = j$; and it is in $S_{<j>}$ iff $a \bmod n_2 = j$. The search space for successful instantiations of rule 3*a* is now reduced to $F_{<i>} \times T_{<i,j>} \times S_{<j>}$. The partitions induced by the constraints are easier to recognize in the following rectified version of rule 3*a*:

4*a*: $q(x, y, z) :- q(u, x, v), q(y, v_1, w), q(w_1, z, t), v = v_1, w = w_1, v^2 \bmod n_1 = i, v_1^2 \bmod n_1 = i, w \bmod n_2 = j, w_1 \bmod n_2 = j.$

Here, the third and the last constraints define $F_{<i>}$ and $S_{<j>}$, respectively. The fourth and fifth constraints together define $T_{<i,j>}$. Since in rectified rules different functions can be specified on the variables that are same in the original rule, we have more flexibility in partitioning the relations of subgoals. □

The idea is to view the body of a given rule as a join expression in which each ordinary subgoal represents a relation. The workload of evaluating the given rule can be partitioned by first replacing the join expression with a few other join expressions of the same type (corresponding to the bodies of restricted rules) in which each ordinary subgoal represents a fragment of the original relation, and then distributing these join expressions over a set of processors for evaluation. We now formalize the idea.

Definition 3.1: A *partition function* $f(\vec{x})$ over a rule r is a mapping $D(x_1) \times \ldots \times D(x_m) \longrightarrow \mathcal{R}$, where $\vec{x} = \langle x_1, \ldots, x_m \rangle$ is a vector of variables in r, $D(x_i)$ is the domain of x_i, and $\mathcal{R}$, called the *range* of $f(\vec{x})$, is a subrange $[0..n-1]$ of integers for some $n \ge 1$. □

Notice that the range of a partition function is a subrange of integers, so it is easy to assign restricted rules to processors. Also, in practice, partition functions are efficient to evaluate

because they are usually simple and evaluated only after their variables are instantiated.

Example 3.2: Consider the following program $\mathcal{P}$.

$5a$: $q(x, y, z)$:- $q(u, x, v)$, $q(y, v_1, w)$, $q(w_1, z, t)$, $v = v_1$, $w = w_1$.
$5b$: $q(x, y, z)$:- $e(x, y, z)$.

Assume the database domain is the integers. Then $f(u, x, v_1, z) \stackrel{\text{def}}{=\!=} [(u + v_1)^2 + (x + z)^2] \bmod 5$ is a partition function over rule $5a$, where the range of the function is $\{0, 1, 2, 3, 4\}$. □

Definition 3.2: Let $\vec{g} = \langle f_1(\vec{x}_1), \ldots, f_k(\vec{x}_k)\rangle$, where $k \geq 0$, be a vector of partition functions over a rule r with ranges $\mathcal{R}_1, \ldots, \mathcal{R}_k$, respectively. The *domain* of $\vec{g}$ denoted by $Dom(\vec{g})$, is the cross product $\mathcal{R}_1 \times \cdots \times \mathcal{R}_k$. For each vector $\vec{v} = \langle v_1, \ldots, v_k\rangle \in Dom(\vec{g})$, a *restricted rule* of r wrt $\vec{g}$ is obtained by appending the conjunction $f_1(\vec{x}_1) = v_1, \ldots, f_k(\vec{x}_k) = v_k$ to the body of r. $\vec{g}$ and $\vec{v}$ are called the *function vector* and the *processing vector*, respectively. Those constraints of r that are in the conjunction are called *partition constraints*, and the other constraints are called *original constraints*. The set of all restricted rules of r obtained from processing vectors in $Dom(\vec{g})$ is denoted by $Comp(r, \vec{g})$. □

By Definition 3.2, if $\vec{g} = \langle\rangle$, that is, the function vector is empty, the only restricted rule of r wrt $\vec{g}$ is r itself.

Example 3.3: (*continued from Example 3.2*). Suppose partition functions $f_1(v) \stackrel{\text{def}}{=\!=} v^2 \bmod 2$, $f_2(v_1) \stackrel{\text{def}}{=\!=} v_1^2 \bmod 2$, $f_3(w) \stackrel{\text{def}}{=\!=} w \bmod 3$, and $f_4(w_1) \stackrel{\text{def}}{=\!=} w_1^2 \bmod 3$ are defined over rule $5a$, and no partition function is defined over rule $5b$. The function vector over $5a$ is $\vec{g}_1 = \langle f_1(v), f_2(v_1), f_3(w), f_4(w_1)\rangle$, and that over $5b$ is $\vec{g}_2\langle\rangle$. Hence $Dom(\vec{g}_1 = \{\langle i, k, j, h\rangle \mid 0 \leq i, k \leq 1$ and $0 \leq j, h \leq 2$ are integers$\}$, $Comp(5a, \vec{g}_1) = \{\mu_{i,k,j,h} \mid 0 \leq i, k \leq 1$ and $0 \leq j, h \leq 2$ are integers$\}$, where $\mu_{i,k,j,h}$ is of the form

$$q(x, y, z) \text{ :- } q(u, x, v), q(y, v_1, w), q(w_1, z, t), v = v_1, w = w_1, f_1(v) = i, f_2(v_1) = k, f_3(w) = j, f_4(w_1) = h.$$

and $Comp(5b, \vec{g}_2)$ contains only rule $5b$. In the restricted rules, original constraints are $v = v_1$ and $w = w_1$, and partition constraints are $f_1(v) = i, f_2(v_1) = k$, etc. □

Definition 3.3: Let $\mathcal{P}$ be a program consisting of rules $r_1, \ldots, r_M$. A *partition scheme* of $\mathcal{P}$ is a set $\mathcal{G} = \{\vec{g}_1, \ldots, \vec{g}_M\}$, where $\vec{g}_i$ is a (possibly empty) function vector over r_i. A *restricted program* of $\mathcal{P}$ wrt $\mathcal{G}$ is a non–empty set containing at most one restricted rule of r_i, for every $1 \leq i \leq M$. A *processing scheme* of $\mathcal{P}$ wrt $\mathcal{G}$ is a set of restricted programs $\{P_1, \ldots, P_m\}$ of $\mathcal{P}$ wrt $\mathcal{G}$, where $m \geq |Comp(r_i, \vec{g}_i)|$ for every $1 \leq i \leq M$, such that every restricted rule is in exactly one restricted program. □

Intuitively, a partition scheme describes how the data should be partitioned, and a processing scheme describes how restricted rules are grouped together. The requirement for a restricted program to contain at most one restricted rule of any rule of the program is to simplify the presentation. The requirement for each restricted rule to be in exactly one restricted program is because (1) to have a correct result, each restricted rule must be evaluated at some processor; and (2) to avoid redundancy, a restricted rule should not be evaluated at more than one processor.

Example 3.4: (*continued from Example 3.3*). The partition scheme is $G = \{\vec{g}_1, \vec{g}_2\}$. One possible processing scheme of $\mathcal{P}$ wrt $\mathcal{G}$ consists of single–rule restricted programs only, where each rule in $Comp(5a, \vec{g}_1)$ and $Comp(5b, \vec{g}_2)$ is a restricted program by itself. Another possible processing scheme of $\mathcal{P}$ wrt $\mathcal{G}$ is obtained by first forming a single–rule program for each restricted rule $\mu_{0,0,0,0}$, $\mu_{0,0,0,1}$, etc., and then placing rule $5b$ into any one of the single–rule programs to form a two–rule program. □

Partition constraints are more general than restricting predicates [13] and discriminating predicates [6] since each rule, by definition, can have only one function in the restricting or discriminating predicate but can have an arbitrary number of partition functions. This not only allows flexibility in specifying data partition (see Section 6) but also permits more efficient parallel evaluation of programs (see Section 4).

The following theorem, used to prove the correctness and the efficiency of the parallel evaluation algorithm in Section 5, indicates that the meaning of a given Datalog program is captured by its processing scheme. Interested readers may refer to [15] for a proof.

Theorem 3.1: Let $\mathcal{P}$ be a program and $\{\mathcal{P}_1, \ldots, \mathcal{P}_m\}$ be a processing scheme of $\mathcal{P}$ wrt a given partition scheme. For any EDB, $lfp(\mathcal{P}) \equiv lfp(\mathcal{P}_1 \cup \ldots \cup \mathcal{P}_m)$. □

IV. A Data Transmission Criterion

We assume that each restricted program is evaluated at exactly one processor using data stored locally. If a tuple t is obtained at processor i and sent to processor j, we call processor i the *sender* and processor j the *receiver* of t.

Conceptually, the evaluation of a restricted program at a processor is a search for all successful rule instantiations of the restricted program that can be obtained from data in the local database. Since data received from other processors are placed into the local database, they allow more successful rule instantiations to be found, but also increase the time needed for the evaluation. Data transmission criteria are used to reduce the amount of data transmitted. The reduction is possible because some tuples computed at a processor may not be needed at other processors. Hence each sender needs to test the data before actually sending them out so that only those data satisfying the data transmission criterion are sent. A good data transmission criterion should satisfy the following requirements:

R1: The test of data using the criterion is based solely on the information locally available to the sender. Such information typically includes the processing scheme, the location of restricted programs, the data stored at the

sender, and the specification of the database domain. Since the purpose of the test is to reduce the cost of data transmission, it is reasonable to require that the test itself does not incur communication overhead.

R2: The criterion should guarantee that even if a tuple fails the test (consequently, it is not sent to a processor) the final result is still the same as if it is sent to the processor. Because (1) many algorithms test data at some specific step prior to the data transmission, (2) it is undesirable to test a tuple more than once wrt a given processor, and (3) the test is intended to improve the efficiency, the correct result of the evaluation should be preserved.

R3: The criterion is nontrivial, that is, it indeed reduces the cost of data transmission as compared to that of broadcasting, at least in some situations.

R4: The criterion can be used efficiently to test the data (to be discussed later in this section).

Intuitively, a tuple t should be sent from a sender to a receiver if it will contribute to the evaluation at the receiver. Such a contribution can be recognized if in the local database of the receiver, a set of tuples, S, can be found so that they and t together form a part of a successful rule instantiation. Thus a data transmission criterion needs to specify which part of a rule needs to be considered and in which range tuples in S are searched. The following definitions provide a way of specifying such information.

Definition 4.1: A constraint (respectively, a partition function) in a restricted rule r *involves* an ordinary subgoal of r if the constraint (respectively, the partition function) and the subgoal share a variable. Let Λ be a set of constraints in a restricted rule r. Two ordinary subgoals (not necessarily distinct) of r, say a and b, are *connected* wrt Λ iff

1) a and b are the same subgoal, or
2) a constraint in Λ involves both a and b, or
3) there is some ordinary subgoal c of r such that a and c are connected wrt Λ and c and b are connected wrt Λ. □

Definition 4.2: A *(connected) component* of a restricted rule r wrt a set of constraints Λ in r is the conjunction of

1) a set of all ordinary subgoals of r that are connected with each other wrt Λ, and
2) all constraints in Λ that involve the ordinary subgoals in the set in 1). □

Example 4.1: Consider the following rule:

$$q(x, y, z) :- q(u_1, x, u_2), a(u_3, y), b(u_4, u_5), q(u_6, z, u_7), u_3 = u_6, u_5 = u_7, u_2 \le u_1, (u_1 + u_3) \bmod 6 = 4, (u_6 + z) \bmod 5 = 3.$$

Constraint $u_3 = u_6$ involves ordinary subgoals $a(u_3, y)$ and $q(u_6, z, u_7)$. If the constraint is in Λ, the two ordinary subgoals are connected. Similarly, constraint $(u_1 + u_3) \bmod 6 = 4$ involves ordinary subgoals $q(u_1, x, u_2)$ and $a(u_3, y)$. If the set of given constraints is $\Lambda = \{(u_1 + u_3) \bmod 6 = 4, u_2 \le u_1, (u_6 + z) \bmod 5 = 3\}$, there are three components wrt Λ: $[q(u_1, x, u_2), a(u_3, y), (u_1 + u_3) \bmod 6 = 4, u_2 \le u_1]$, $[b(u_4, u_5)]$, and $[q(u_6, z, u_7), (u_6 + z) \bmod 5 = 3]$. Notice that in this example, if $\Lambda = \emptyset$, each ordinary subgoal is a component by itself, while if Λ contains every constraint, the entire rule body is a single component. □

Definition 4.3: For a given program and an EDB, a *least fixpoint relation* of an ordinary subgoal in a rule is a relation of the predicate of the subgoal that contains all tuples of the predicate derivable from the EDB using the given program. □

We now consider a nontrivial data transmission criterion.

Definition 4.4: A *potential usefulness criterion* (thereafter, a *potential usefulness*) wrt an ordinary subgoal b of a restricted rule r is a triple (Λ, Φ, Γ), where Λ contains all partition constraints in r, Φ contains a least fixpoint relation of each ordinary subgoal in r, and Γ contains the only component of r wrt Λ that contains b. □

Definition 4.5: Let $C = c_1, \ldots, c_m, L$ be a component of a restricted rule r wrt a set of constraints in r, where c_is are ordinary subgoals and L is the conjunction of constraints. Let Φ contains a relation R_i for each c_i. A substitution θ *satisfies* C wrt Φ if all constraints in $L\theta$ are true, and each $c_i\theta$ is a tuple in R_i. □

Definition 4.6: A tuple t *satisfies* a potential usefulness criterion (Λ, Φ, Γ) wrt an ordinary subgoal b of a restricted rule r if there is a substitution θ satisfying the component in Γ wrt Φ so that $b\theta$ is t. □

Example 4.2: Consider the following restricted rule:

$$q(x, y, z) :- q(u_1, x, u_2), a(u_3, y), b(u_4, z, u_5), u_3 = u_4, u_1 = u_5, (u_1 + y) \bmod 3 = 1.$$

Assume that $\Lambda = \{(u_1 + y) \bmod 3 = 1\}$; Φ contains $a(7, 4)$ and $a(3, 6)$ for a, $b(7, 8, 3)$ for b, and a least fixpoint relation for q; and Γ contains $\{q(u_1, x, u_2), a(u_3, y), (u_1 + y) \bmod 3 = 1\}$. (Λ, Φ, Γ) is a potential usefulness criterion wrt $q(u_1, x, u_2)$, and both $q(3, 5, 6)$ and $q(4, 2, 3)$ satisfy the criterion. Notice that $q(4, 2, 3)$ is not an instantiation of $q(u_1, x, u_2)$ in any successful instantiation of the rule wrt the database (Φ). □

Notice that although a tuple satisfying the potential usefulness wrt a subgoal of a restricted rule may or may not appear as the instantiation of the subgoal in any successful instantiation of the rule, a tuple that does not satisfy the potential usefulness wrt any subgoal of a restricted rule will never appear in any successful instantiation of the rule. To prevent the sender from evaluating r, which is supposed to be done at the receiver, only partition constraints are included in Λ of the potential usefulness.[2] The potential usefulness may not be a good data transmission criterion for arbitrary processing schemes for two reasons. First, the test may not be feasible because Φ contains least fixpoint relations of IDB predicates which may not be completely evaluated during the evaluation. Second, the

[2] In practice, Λ can also contain original constraints that involve no more than one ordinary subgoal. This will reduce the cost of the data transmission but will not increase the time needed for the test.

time needed for the test may depend on the size of relations in Φ. As a result, both requirements R1 and R2 may be violated. These problems are caused by global partition constraints (or functions) defined as follows.

Definition 4.7: A constraint (respectively, partition function) in a rule r is *global* if it involves at least two ordinary subgoals. Otherwise, it is *local*. □

For example, the function $f(u, x, v_1, z)$ in Example 3.2 is global, while all functions in Example 3.3 are local. In a restricted rule, a global constraint corresponds to a multiway join condition in the relational algebra. To find a substitution satisfying a component of the rule that contains global constraints is to find a tuple in the join where the join conditions correspond to the global constraints, and each relation corresponds to an ordinary subgoal in the component. This join is impossible at the sender unless all relations involved are already in the local database. The cost of this join depends on the size of the relations involved. Furthermore, if the relations involved are associated with IDB predicates, an empty result of the join may not necessarily provide the basis for not sending a tuple. If no global partition constraint is in the rule, Γ will contains exactly one ordinary subgoal. As a result, a tuple satisfies the potential usefulness wrt a subgoal b iff it is a tuple of b and satisfies the set of partition constraints local to b. Notice that the only effect that local partition constraints have on a subgoal is to partition the relation of the subgoal into a set of fragments. Since the partition scheme is known to all processors, with only local partition constraints, to test if a tuple satisfies the potential usefulness wrt a subgoal b of a rule r at a processor s, a sender only needs to check if the tuple belongs to the fragment of b's relation assigned to s. It is obvious that the test of data using the potential usefulness is always feasible in this case. Because of this observation our strategy is called the data partition strategy. The following proposition indicates that using the potential usefulness with local constraints is also efficient.

Proposition 4.1: If the partition scheme contains only local partition functions whose evaluation on constants take a constant time, then whether a tuple satisfies the potential usefulness wrt a subgoal of a restricted rule can be tested in time $O(k)$, if k partition functions are defined over the subgoal.

Proof. Because partition functions are local, the time needed for the test is that needed to evaluate the k partition functions that are defined over the subgoal. By assumption, each partition function can be computed in a constant time. Thus the proposition is proved. □

The potential usefulness is a good data transmission criterion for processing schemes that contains only local partition constraints (that is, it satisfies all four requirements listed at the beginning of this section). Local partition constraints present no limit to specifications of the data partition in practice, and drastically simplify the test of data using the potential usefulness.

Definition 4.8: A partition scheme is *local* if it contains only local partition functions. A processing scheme is *local* if it is with respect to a local partition scheme. □

In the remainder of this paper, we consider only local partition schemes.

Algorithm A1: Parallel semi-naive evaluation

```
1.  Initialization;
2.  Sending;
3.  Receiving;
4.  Decomposition;
5.  Accumulation;
6.  REPEAT
7.      REPEAT
8.          Evaluation;
9.          Sending;
10.         Receiving;
11.         Decomposition;
12.         Accumulation
13.     UNTIL ΔP' = θ for every IDB predicate p;
14.     Finish : = Termination;
15.     IF NOT Finish THEN BEGIN
16.         Wait for data from other processors;
17.         Receiving;
18.         Decomposition;
19.         Accumulation
20.     END
21. UNTIL Finish;
```

Fig. 1. A parallel semi–naive evaluation algorithm.

V. An Evaluation Algorithm

In this section, we present an algorithm which is a modification of the parallel semi–naive algorithm in [13]. The main differences between our algorithm and the one in [13] are the following:

1) Our algorithm is based on the partitioning of data. In the evaluation of a restricted rule, each ordinary subgoal is associated with tuples that are available at the processor and satisfy the potential usefulness wrt that subgoal. This set of tuples is referred to as the *input relation* of the subgoal. On the contrary, in the algorithm in [13], each ordinary subgoal with a predicate p is associated with a set of all tuples of p that are available at the processor, of which the input relation is a subset.
2) Our algorithm uses the potential usefulness as the data transmission criterion, while the algorithm in [13] does not.
3) As a result of the previous two differences, our algorithm tends to do less amount of work (see Proposition 5.3 and the discussion thereafter).

The algorithm is given in Fig. 1. The procedures and the function in the algorithm perform the following tasks. *Initialization* computes the initial set of tuples for every IDB predicate by evaluating exit rules. *Evaluation* computes new tuples for each IDB predicate, where new means that the tuples are not yet in the relation of the predicate in the local database. *Sending* transmits new tuples so that each tuple being sent satisfies the potential usefulness wrt some ordinary subgoal of some restricted rule at its receivers. *Receiving* collects new tuples that are received during the most recent iteration. *Decomposition* assigns newly received tuples to relations of subgoals based on the potential usefulness wrt the subgoals. *Accumulation* updates relations of predicates by adding to them tuples that are either obtained locally or received from other processors. *Termination* evaluates to TRUE if no processor is busy and no data is sent and yet not received. Methods similar to those discussed in [13], [6] can be used to implement this function. The details of the six procedures are given in the Appendix. The parallel evaluation of a local processing scheme of a program is defined as the following.

Definition 5.1: Let $\{\mathcal{P}_1, \ldots, \mathcal{P}_m\}$ be a local processing scheme of a program $\mathcal{P}$. The *parallel evaluation* of the processing scheme is to evaluate each $\mathcal{P}_i$ at a distinct processor, say s_i, based on Algorithm A1. The *result of the parallel evaluation* of the processing scheme is the set of all EDB and IDB tuples at the m processors at the end of the parallel evaluation. □

We assume the EDB is initially distributed so that a tuple t is placed in the input relation of a subgoal b_j at a processor s iff t satisfies the potential usefulness wrt b_j. We also assume there is a final step to collect tuples in the result of the parallel evaluation. The following theorem states the correctness of the parallel evaluation. The proof is in the Appendix.

Theorem 5.1: Let $\{\mathcal{P}_1, \ldots, \mathcal{P}_m\}$ be a local processing scheme of a program $\mathcal{P}$. For any given EDB, a tuple is in $lfp(\mathcal{P})$ iff it is in the result of the parallel evaluation of $\{\mathcal{P}_1, \ldots, \mathcal{P}_m\}$.□

The performance of the algorithm is characterized by the following propositions whose proofs are in the Appendix.

Proposition 5.2: In Algorithm A1, a tuple is tested in procedure *Sending* for satisfying the potential usefulness wrt any subgoal that has a predicate p_i only if it is in the incremental relation ΔP_i of p_i returned by procedure *Initialization* or *Evaluation*. Furthermore, it is tested exactly once wrt each subgoal that has the predicate p_i. □

To implement Algorithm A1, the test of data using the potential usefulness in procedure *Decomposition*, at lines 4, 11, and 18, can be avoided by attaching a flag to each tuple to indicate subgoals wrt which the tuple satisfies the potential usefulness. Thus *Decomposition* only needs to add received tuples to input relations and incremental input relations according to the attached flags. As a result, the test in *Sending* is the only test needed.

Proposition 5.3 Let $\{\mathcal{P}_1, \ldots, \mathcal{P}_m\}$ be a local processing scheme of a program $\mathcal{P}$. For any given EDB, if a rule in $\mathcal{P}$ has a non–empty partition vector and is either an exit rule or a rule having two or more ordinary subgoals, then each successful instantiation of this rule is obtained exactly once in the parallel evaluation of the processing scheme. □

Generally speaking, the workload of evaluating a given program can be characterized by the set of successful rule instantiations obtained in the evaluation. Previous methods attempt to partition this workload by partitioning rule instantiations directly. But rule instantiations are not stored explicitly. Thus to decide whether a tuple should be sent from a processor to another, rule instantiations assigned to the receiver may have to be obtained at the sender so that it becomes possible to test whether the tuple appears in these rule instantiations. Since several processors may try to send the same tuple at the same time and a rule instantiation may be needed to test different tuples at a processor at different times, the same rule instantiation may have to be obtained many times at various processors. Our approach, on the other hand, partitions the workload by partitioning relations in the database into fragments (that is, the input relations) and allocateing these fragments to processors in such a way that each successful rule instantiation is obtained at precisely one processor. Thus to decide whether a tuple should be sent to a processor, all we have to do is to see if the tuple belongs to a fragment allocated to the receiver. As a result, we obtain the performance gain indicated by Proposition 5.3.

VI. Design of Schemes

In this section, we consider the design of a local processing scheme for a given program. Given a program $\mathcal{P}$ and $\mathcal{N}$ processors, the task is to design a local partition scheme G and a processing scheme $\{\mathcal{P}_1, \ldots, \mathcal{P}_m\}$ wrt G so that the number of restricted programs, that is, m, is small enough to assign no more than one restricted program to each processor and large enough to utilize as many processors as possible. The partition scheme should also satisfy other requirements, such as to partition large relations instead of small ones. Because of these requirements, the design is nontrivial. We identify three major design steps.

Choosing the format of a partition scheme: For each rule, decide subgoals to be partitioned, partition functions needed for chosen subgoals, variables used and operations performed by partition functions, and relationships among partition functions.

Determining ranges of partition functions: Find a range for each partition function based on the number of processors, the importance of the partition function, and the relationships among partition functions.

Designing a processing scheme: Find processing vectors based on the partition scheme, and create restricted programs.

We now illustrate the design steps with an example.

Example 6.1: Consider the following rectified program:

7a: $q(x,y,z)$:- $b(y, v_1, w_1)$, $q(u_1, x, v_2)$, $q(u_2, u_3, w_2)$, $t(z, v_3)$, $v_1 = v_2$, $w_1 = w_2$, $u_1 = u_2$, $v_2 = v_3$, $u_2 = u_3$.

7b: $q(x,y,z)$:- $c(w_3, v_4, z)$, $q(u_4, v_5, y)$, $d(z, w_4, u_5)$, $w_3 = w_4$, $v_4 = v_5$, $u_4 = u_5$.

7c: $q(x, y, z)$:- $e(x, y, z)$.

Assume six processors are available. In the following, decisions are made without explanation (which will become clear in subsequent subsections). First, the following format of a partition scheme is chosen where n_is are the yet to be determined ranges of partition functions.

$f_1(u_1) \stackrel{\text{def}}{=\!=} u_1 \bmod n_1$ on $q(u_1,x,v_2)$ in rule 7*a*,

$f_2(v_2) \stackrel{\text{def}}{=\!=} v_2^2 \bmod n_2$ on $q(u_1,x,v_2)$ in rule 7*a*,

$f_3(u_2) \stackrel{\text{def}}{=\!=} u_2 \bmod n_1$ on $q(u_2,u_3,w_2)$ in rule 7*a*,

$f_4(w_3,v_4,z) \stackrel{\text{def}}{=\!=} (w_3^2 + v_4^2 + z^2) \bmod n_4$ on $c(w_3,v_4,z)$ in rule 7*b*,

$f_5(u_4) \stackrel{\text{def}}{=\!=} u_4 \bmod n_5$ on $q(u_4,v_5,y)$ in rule 7*b*,

$f_6(x,y,z) \stackrel{\text{def}}{=\!=} (x+y+z) \bmod n_6$ on $e(x,y,z)$ in rule 7*c*.

Notice that f_1 and f_3 (over 7*a*) are the same mapping (on different variables). Let the partition scheme be $\mathcal{G} = \{\vec{g}_1, \vec{g}_2, \vec{g}_3\}$, where $\vec{g}_1 = \langle f_1, f_2, f_3\rangle$, $\vec{g}_2 = \langle f_4, f_5\rangle$, and $\vec{g}_3 = \langle f_6\rangle$. Next, let $n_1 = 3$, $n_2 = 2$, $n_4 = 2$, $n_5 = 3$, and $n_6 = 6$. Since f_1 is defined on variable u_1, f_3 on u_2, and $u_1 = u_2$, the following tables give all useful processing vectors of rules (that is, they generate restricted rules that will have a successful instantiation for at least one EDB).

	f_1	f_2	f_3		f_4	f_5		f_6
	0	0	0		0	0		0
	0	1	0		1	0		1
For rule 7*a*	1	0	1	For rule 7*b*	0	1	For rule 7*c*	2
	1	1	1		1	1		3
	2	0	2		0	2		4
	2	1	2		1	2		5

A processing scheme can contain six restricted programs $\mathcal{P}_1, \ldots, \mathcal{P}_6$, where $\mathcal{P}_i$ is obtained from row i of the three tables. For example, $\mathcal{P}_3$ is

$q(x, y, z)$:- $b(y, v_1, w_1)$, $q(u_1, x, v_2)$, $q(u_2, u_3, w_2)$, $t(z, v_3)$,
$v_1 = v_2$, $w_1 = w_2$, $u_1 = u_2$, $v_2 = v_3$, $u_2 = u_3$, $f_1(u_1) = 1$,
$f_2(v_2) = 0$, $f_3(u_2) = 1$.
$q(x, y, z)$:- $c(w_3, v_4, z)$, $q(u_4, v_5, y)$, $d(z, w_4, u_5)$, $w_3 = w_4$,
$v_4 = v_5$, $u_4 = u_5$, $f_4(w_3, v_4, z) = 0$, $f_5(u_4) = 1$.
$q(x, y, z)$:- $e(x, y, z)$, $f_6(x + y + z) = 2$. □

6.1 The Format of A Partition Scheme

The following aspects are important in determining the format of a partition scheme.

Ordinary Subgoals to be Partitioned: Heuristics can be applied to determine subgoals to whom partition functions should be assigned. One heuristic is to partition subgoals whose predicates are associated with large relations so as to reduce the cost of evaluating rules. The size of IDB relations may be estimated [4] or obtained from statistical data. A second heuristic is based on the syntax of the program, for instance, choosing subgoals according to the pivoting property to eliminate the data transmission. Yet another heuristic is to choose subgoals that share some common property. For example, to choose subgoals that share variables (in the original, not–yet–rectified program) is likely to reduce the total number of useful processing vectors of a rule.

Number of Partition Functions: There are two reasons for specifying multiple partition functions on a chosen subgoal. First, if the subgoal relates to other subgoals via its join variables (that is, those variables related to variables in other subgoals through equality), one function can be defined for each join variable. Secondly, it can impose a "composite hash partition" to deal with data skew problems [10]. On the other hand, specifying too many partition functions may make many of them void, that is, their ranges become one.

Arguments and Operations of Partition Functions: The choice of arguments and operations of partition functions can be based on two heuristics. The first is to partition each relation into fragments of approximately equal size so as to balance the workload among processors. The second is to enforce certain dependencies among partition functions (see next paragraph) to reduce the number of restricted programs and the cost of the data transmission.

Dependencies of Partition Functions: A type of relationship among partition functions is defined as follows.

Definition 6.1: Let b and b' be ordinary subgoals in a restricted program, and $f(x_1, \ldots, x_k)$ and $g(y_1, \ldots, y_k)$ be partition functions local to b and b', respectively. $f(x_1, \ldots, x_k)$ and $g(y_1, \ldots, y_k)$ are *equivalent* if for every constant vector $\langle a_1, \ldots, a_k\rangle$, $f(a_1, \ldots, a_k) = g(a_1, \ldots, a_k)$; they are *join related* if 1) they are equivalent, 2) b and b' are different subgoals in the same rule, and 3) for $1 \le i \le k$, x_i and y_i are join variables. □

The join related dependency imposes an equivalence relation, on partition functions over a rule, whose equivalence classes, called *join classes*, can be used to determine ranges of partition functions and to eliminate useless processing vectors. In Example 6.1, functions $f_1(u_1)$ and $f_3(u_2)$ are join related. Because of this, only six processing vectors are generated. The other 12 possible processing vectors are useless.

6.2 Ranges of Partition Functions

Ranges of partition functions are determined on the rule–by–rule basis. Given $\mathcal{N}$ processors and a function vector $\langle f_1, \ldots, f_k\rangle$ over a rule r. Let n_i be the size of the range of f_i. Since each restricted rule of r is determined by a processing vector $\langle v_1, \ldots, v_k\rangle$, where $0 \le v_i < n_i - 1$, there are total $n_1 \times \ldots \times n_k$ restricted rules of r. When there are join related dependencies, partition functions in the same join class must have the same range and be evaluated to the same value in each restricted rule of r. Suppose $f_1, \ldots, f_k$ are partitioned into m join classes. If partition functions in the join class i have the range x_i, the total number of useful processing vectors becomes $x_1 \times \ldots \times x_m$. It is, therefore, sufficient to determine ranges for join classes rather than for individual partition functions. The problem is then to determine integer values of x_i, $1 \le i \le m$, such that,

1) $x_i \ge 1$.
2) $\prod_{i=1}^{m} x_i \le \mathcal{N}$ (that is, a processor evaluates no more than

one restricted rule of r).

3) $\mathcal{N} - \prod_{i=1}^{m} x_i$ is minimized (for maximum utilization of processors).
4) The number of x_i that is less than two is minimized (for maximum utilization of partition functions).

Notice that Conditions 1 and 2 together with one of Conditions 3 and 4 can always be satisfied simultaneously, but the four conditions together may not.[3]

Example 6.2: Assume $\mathcal{N} = 20$ and $m = 5$. An assignment that satisfies Conditions 1, 2, and 4 is: $x_1 = 2$, $x_2 = 2$, $x_3 = 2$, $x_4 = 2$, and $x_5 = 1$. But the product of x_is is only 16. An assignment that satisfies Condition 1, 2, and 3 is: $x_1 = 2$, $x_2 = 2$, $x_3 = 5$, $x_4 = 1$, and $x_5 = 1$. But two instead of one x variables are assigned value 1. In fact, for this example, any assignment satisfying Conditions 1, 2, and 4 must have a product of x values that is not greater than 16; and any assignment satisfying Conditions 1, 2, and 3 must have at least two x variables with value 1. Thus no assignment can satisfy all four conditions. □

Algorithm A2: Even assignment of ranges.

Input: The total number of processors $\mathcal{N}$ and the total number of join classes m.

Output: Ranges of join classes $x_1, \ldots, x_m$ satisfying Conditions 1, 2 and 4.

Method:

```
d := ⌊2^(log2 N/m)⌋;
FOR i = 1 TO m DO
   x_i := d;
K := d^m;
C := (d+1)/d;
FOR i := 1 TO m DO
   IF K × C ≤ N
      THEN BEGIN
         K:= K × C;
         x_i := x_i + 1;
      END
   ELSE RETURN(x_1, ..., x_m);
```

Fig. 2. An algorithm for even assignment of ranges

In the sequel, due to the limited space, we present only one method for finding values of x_is. The method is given in Fig. 2 For any $\mathcal{N} \geq 1$, we have $d^m \leq \mathcal{N} \leq (d+1)$, where $d \geq 1$ is an integer. Thus the value of each x_i is either d or $d+1$. The method first assigns the value d to each x_i and then adds one to each variable in turn until the product of x_is becomes as close to $\mathcal{N}$ as possible yet still no more than $\mathcal{N}$. The algorithm is very efficient. Applying Algorithm A2 to Example 6.2, the assignment that satisfies Conditions 1, 2, and 4 is obtained. It can also be shown that although in the worst case, one half of processors are not utilized to evaluate the rule, the utilization of processors is good when m is small or $\mathcal{N}$is large.

Once values of x_is are determined, they are assigned to join classes. When all partition functions are completely specified, the partition scheme is then the set of function vectors each of which is a (possibly empty) sequence of the non-void partition functions over a rule.

6.3 Grouping Processing Vectors

To obtain restricted programs, for each rule that has a non-empty function vector, all processing vect0rs are generated and placed in a *processing vector table* which is defined as follows:

Definiton 6.2: Let $\vec{\mathbf{g}} = \langle f_1, \ldots, f_m \rangle$ be a function vector over a rule r, with ranges $n_1, \ldots, n_m$ where $n_i > 1$. A *processing vector table*, T_r of r (wrt $\vec{\mathbf{g}}$) has columns $c_1, \ldots, c_m$, and each row of the table is a processing vector $\langle v_1, \ldots, v_m \rangle$ in $Dom(\vec{\mathbf{g}})$ such that $v_i = v_j$ if f_i and f_j are in the same join class. Partition function f_i is said to be the function of the column c_i and of the element v_i, $1 \leq i \leq m$. □

Algorithm A3: Grouping processing vectors

Input: A set of processing vector tables $T_1, \ldots, T_M$.

Output: A set of groups of processing vectors $W_1, \ldots, W_j$, where j is less than or equal to the number of processors, each W_i contains at most one processing vector from each processing vector table, and every processing vector in processing vector tables is in one W.

Method:

```
1   j := 0;
2   WHILE not all processing vector
    tables are empty DO BEGIN
3    j := j+1;
4    W_j := ∅;
5    FOR each non-empty processing
      vector table T_r DO BEGIN
6      Select the first vector v⃗ from
        T_r;
7      W_j := W_j ∪ {v⃗ with label r};
8      T_r := T_r − {v⃗ }
9    END;
10  END;
11  RETURN W_1, ..., W_j;
```

Fig. 3. A simple algorithm for grouping processing vectors.

Since ranges of partition functions are decided individually for each rule, the sizes of processing vector tables are not necessarily equal.

To construct restricted programs, the processing vectors in

[3] To the best of our knowledge, this is not a known linear or nonlinear programming problem.

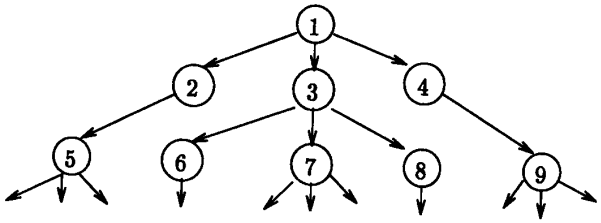


Fig. 4. The tree of the base relations.

processing vector tables should be grouped so that in any group, each processing vector is from a different processing vector table. Fig. 3 gives an algorithm for doing that. With each group of processing vectors W_k returned by Algorithm A3, a restricted program $\mathcal{P}_k$ is constructed so that for each (labeled) processing vector $r : v$ in W_k, $\mathcal{P}_k$ contains the restricted rule of r defined by v. Rules in $\mathcal{P}$ that have empty function vectors are either added to existing restricted programs or used to form restricted programs by themselves, as long as the total number of restricted programs does not exceed the total number of processors.

Example 6.3: The tables given in Example 6.1 are the processing vector tables. The groups of processing vectors returned by Algorithm A3 are

$$
\begin{aligned}
W_1 &= \{7a\text{: }\langle 0, 0, 0\rangle, 7b\text{: }\langle 0, 0\rangle, 7c\text{: }\langle 0\rangle\};\\
W_2 &= \{7a\text{: }\langle 0, 1, 0\rangle, 7b\text{: }\langle 0, 1\rangle, 7c\text{: }\langle 1\rangle\};\\
W_3 &= \{7a\text{: }\langle 1, 0, 1\rangle, 7b\text{: }\langle 1, 0\rangle, 7c\text{: }\langle 2\rangle\};\\
W_4 &= \{7a\text{: }\langle 1, 1, 1\rangle, 7b\text{: }\langle 1, 1\rangle, 7c\text{: }\langle 3\rangle\};\\
W_5 &= \{7a\text{: }\langle 2, 0, 2\rangle, 7b\text{: }\langle 0, 2\rangle, 7c\text{: }\langle 4\rangle\};\\
W_6 &= \{7a\text{: }\langle 2, 1, 2\rangle, 7b\text{: }\langle 1, 2\rangle, 7c\text{: }\langle 5\rangle\}.
\end{aligned}
$$

The restricted program corresponding to W_3 is $\mathcal{P}_3$ shown in Example 6.1. Notice that $f_3(u_2)$ and $f_5(u_4)$ define the same partition on relation of q. But subgoals $q(u_2, u_3, w_2)$ and $q(u_4, v_5, y)$ in $\mathcal{P}_3$ use different fragments of q, as indicated by processing vectors. Thus the processor evaluating $\mathcal{P}_3$ must store data from two fragments of q. If $7b : \langle 0, 1\rangle$ in W_2 and $7b : \langle 1, 0\rangle$ in W_3 are exchanged, the two subgoals will share the same fragment, thus reducing the cost of data transmission. □

As shown in Example 6.3, the way in which processing vectors are grouped together to form restricted programs affects the cost of data transmission. An optimization goal can be to minimize the number of distinct fragments stored at processors. Due to limited space, this subject will not be dealt with in the current paper.

VII. An Experiment

In this section, we describe an experiment intended for studying the performance of the parallel evaluation. We concentrate on the workload distribution, communication costs, and speedup factors.

The parameters of the experiment include a datalog program, a query, base relations, a system architecture, and the partition/processing schemes. We consider the following program with the query $q(1, y)$ where 1 is a constant.

$$
\begin{aligned}
q(x, y) &:- a(x, z), q(z, w), b(w, y)\\
q(x, y) &:- c(x, y)
\end{aligned}
$$

All three base relations a, b, c are identical and are extracted from the tree in Fig. 4 in which nodes are labeled by natural numbers, and each node labeled by an odd integer has three children and that by an even integer has only one child. For each arc from i to j, (i, j) is a tuple of the relation. Clearly, the data distribution in this database is not balanced. The experiment is performed on a shared-nothing architecture consisting of four processors each of which is a SUN workstation. Direct communication between any two processors is achieved by passing messages over a network. The evaluation cost is measured by the number of tuples involved in joins and in communication since it is proportional to the actual amount of CPU and I/O time.

Two partitioning strategies are considered. In the first strategy, only base relations a and c re partitioned, and each processor p_i, $i = 0, 1, 2, 3$, evaluates the restricted program

$$
\begin{aligned}
q(x, y) &:- a(x, z), q(z, w), b(w, y), (x \bmod 4) = i\\
q(x, y) &:- c(x, y), (x \bmod 4) = i
\end{aligned}
$$

Since q is not partitioned, tuples derived at each processor must be sent to all other processors. In the second strategy, relations a, q, b are partitioned, and the restricted programs are of the form

$$
\begin{aligned}
q(x, y) &:- a(x, z), q(z, w), b(w, y), (z \bmod 2) =\\
&\quad j, (w \bmod 2) = k\\
q(x, y) &:- c(x, y)
\end{aligned}
$$

Each processor p_i evaluates the restricted program for which j and k are respectively the first and the second digit of the binary representation of i. Importantly, each derived tuple is sent to no more than one processor.

The workload at each processor is measured by the size of the input to joins performed at the processor. Fig. 5 shows the total number of tuples that are joined at each processor in each iteration for the two partitioning strategies. The workload for the second partitioning strategy is much lower than that for the first one because relation q is partitioned rather than duplicated among processors. As expected, in both cases, the parallel evaluation has achieved a certain degree of workload balancing on the unbalanced input database. In practice, however, estimating distribution and finding a satisfactory method that balances the workload may be difficult.

The communication cost of a processor is measured by the number of tuples that are received and sent by the processor in each iteration. Fig. 6 shows the communication cost of the two strategies. Again, the second partitioning strategy has a much smaller communication cost because each derived tuple of q is sent to no more than one processor.

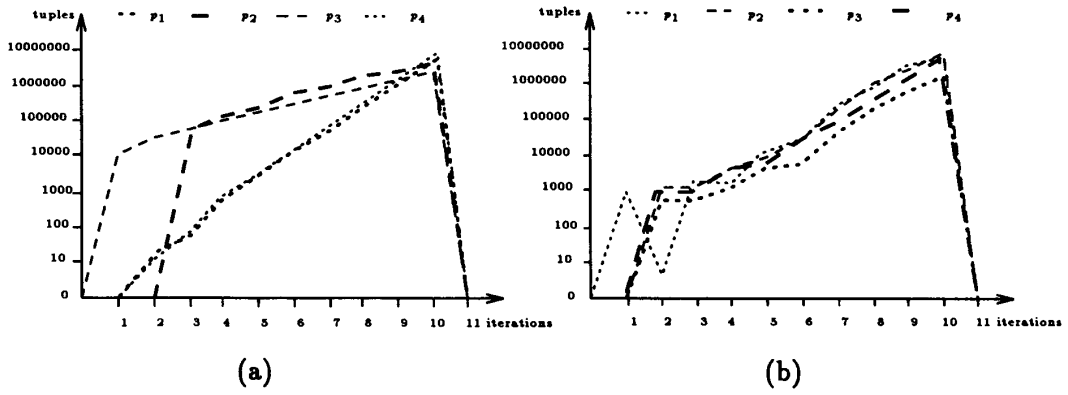


Fig. 5. Workload distribution of the first (a) and the second (b) strategies.

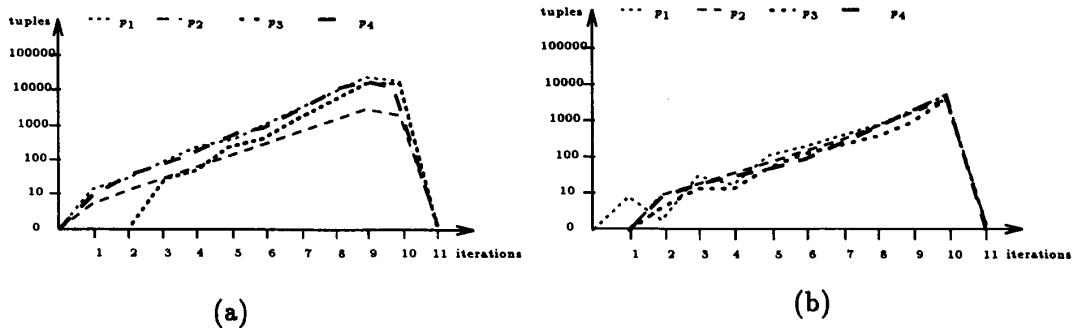


Fig. 6. Communication costs of the first (a) and the second (b) strategies.

To compare the parallel evaluation with the sequential one, we measure the speedup factor which is defined as the ratio s/t, where s is the total number of tuples participated in joins in the sequential evaluation, and $t = max\{t_0, t_1, t_2, t_3\}$ with t_i, $i = 0, 1, 2, 3$, being the total number of tuples participated in joins or are received/sent at processor p_i in the parallel evaluation. In other words, the speedup factor is measured using the most loaded processor. Notice that a linear speedup is possible only if no communication cost is incurred and the workload among processors is perfectly balanced. Speedups of 2.42 and 3.24 were obtained, respectively, for the first and the second partitioning strategies after the first 11 iterations on a sufficiently large input treeThis is considered reasonably good in view of the unbalanced input.

To summarize, the parallel evaluation proposed in this paper shows promising features that improve the sequential evaluation of datalog programs. As the experiment indicated, an important factor related to the performance is the design of a partition scheme that balances workload among processors.

VIII. Conclusion

In this paper, we present a practical parallel evaluation strategy, based on the data partition, to evaluate general Datalog programs. The partition and the processing schemes presented are more general than those of previous strategies. A notion of potential usefulness is used as a criterion that can be efficiently used to reduce the amount of data transmitted. The design of a partition and processing scheme for a given Datalog program is studied. Heuristics and algorithms are provided for the design process. Experiment results are provided to show promising features of our approach.

Similar to [13], our strategy can easily be extended to stratified Datalog programs with negations. Also, definitions of partition constraint can be extended to allow conditions such as $2 \leq u$ and $u \leq 5$. Such conditions can be used to generate equal–size fragments and to identify fragments associated with subgoals of the restricted rules.

The strategy in this paper can also be extended so that it can introduce additional partition functions during the evaluation to divide a restricted program that causes an unbalanced heavy load into two or more new restricted programs and to assign

them to several processors.

Further research is needed to find an efficient algorithm to group processing vectors so that the cost of data transmission and the storage requirement are minimized. A comprehensive study of the performances of ours and other parallel evaluation strategies is also an important direction of future research.

APPENDIX A

A.1. Procedures of Algorithm A1

The following notations are needed. Let $\mathcal{P}$ be a restricted program evaluated at processor s. Denote the IDB predicates in $\mathcal{P}$ by $p_1, \ldots p_K$, the EDB subgoals in $\mathcal{P}$ by $b_1, \ldots, b_M$, and the IDB subgoals in $\mathcal{P}$ by $a_1, \ldots, a_N$. Each IDB predicate p_i is associated with three relations: P_i, an relation containing tuples of p_i accumulated at s through both local computation and data transmission, ΔP_i, an incremental relation containing tuples of p_i that are locally computed in the recent iteration and are not yet in P_i, and $\Delta P'_i$, another incremental relation containing tuples of p_i that are received in the recent iteration and are not yet in P_i. Implicitly, there is also a relation Q for each EDB predicate q. The input relation associated with an ordinary subgoal is denoted by B_i for EDB subgoal b_i and A_j for IDB subgoal a_j. Each IDB subgoal a_j is also associated with an *incremental input relation*, denoted by ΔA_i, containing tuples in the incremental relations of its predicate that satisfy the potential usefulness wrt the subgoal. Operations EVAL–RULE, EVAL, EVAL–RULE–INCR, and EVAL–INCR are similar to those defined in [9] except that the evaluation is based on the input and incremental input relations.

Initialization:
FOR j:= 1 TO N Do
  $A_j := \emptyset$;
FOR every IDB predicate p_i DO
  $\Delta P_i := EVAL(p_i, B_1, \ldots, B_M, \emptyset, \ldots, \emptyset)$;

Evaluation:
FOR every IDB predicate p_i DO BEGIN
  $\Delta P_i :=$ EVAL–INCR ($p_i, B_1, \ldots, B_M$,
    $A_1, \ldots, A_N, \Delta A_1, \ldots, \Delta A_N$);
  $\Delta P_i := \Delta P_i - P_i$;
END;

Sending:
FOR each processor s DO
  $H_s := \emptyset$;
FOR each IDB predicate p_i DO
  FOR each tuple t in ΔP_i DO
    FOR each subgoal a_j with predicate p_i
      DO BEGIN
        $U := \{s \mid s$ is a processor and t satisfies the potential usefulness wrt a_j at $s\}$;
        FOR each $s \in U$ DO
          $H_s := H_s \cup \{t\}$;
      END;
FOR each processor s s.t. $H_s \neq \emptyset$ DO
  Send H_s to s;

Receiving:
FOR every IDB predicate p_i DO BEGIN
  $\Delta P'_i := \{t \mid t$ is a tuple of p_i received during the most recent iteration$\}$;
  $\Delta P'_i := \Delta P'_i - P_i$
END;
Decomposition:
FOR every IDB predicate p_i DO
  FOR every subgoal a_j with predicate p_i DO BEGIN
    $\Delta A_j := \{t \mid t \in \Delta P'_i$ and t satisfies the potential usefulness wrt $a_j\}$;
    $A_j := A_j \cup \Delta A_j$
  END;
Accumulation:
FOR every IDB predicate p_i DO
  $P_i := P_i \cup \Delta P_i \cup \Delta P'_i$;

A.2. Proof of Theorem 5.1

We first provide some concepts on derivation which will be used in the following proof. In this paper, a derivation of a tuple t using a program $\mathcal{P}$ is represented by a *derivation tree*. The root of the derivation tree is (labeled by) t. Each leaf node in the tree is either an EDB tuple or an instantiation of a constraint that is true under the conventional interpretation. There is an internal node g with children nodes $c_1, \ldots, c_n$ in the tree iff $g :- c_1, \ldots, c_n$ is a successful instantiation of a rule r in $\mathcal{P}$ in the derivation. To emphasize that all successful instantiations represented by the derivation tree belong to rules in $\mathcal{P}$, we say that the derivation tree is wrt $\mathcal{P}$. The height of a derivation tree is the number of edges in the longest path from the root to a leaf node.

Proof: By Theorem 3.1, we only need to show that a tuple is in $lfp(P_1 \cup \ldots \cup P_m)$ iff it is in the result of the parallel evaluation.

(*If*) The evaluation at any processor s has the following properties.

1) The evaluation starts at line 1 of A1 in which only input relations of EDB subgoals are used. By definition, the tuples returned by EVAL–RULE are in $lfp(P_1 \cup \ldots \cup P_m)$.
2) Tuples are sent from s to other processors at lines 2 and 9 of A1 only if they are computed at s.
3) Since all processors execute the same algorithm and only procedures *Initialization* and *Evaluation* can generate new tuples, all tuples in input relations and incremental input relations at line 8 of A1 are in $lfp(\mathcal{P}_1 \cup \ldots \cup \mathcal{P}_m)$.
4) Rules evaluated at processor s belong to a single restricted program.

Thus, each processor can only produce a subset of tuples in $lfp(\mathcal{P}_1 \cup \ldots \cup \mathcal{P}_m)$.

(*Only if*) For each tuple in $lfp(\mathcal{P}_1 \cup \ldots \cup \mathcal{P}_m)$, the *minimal derivation tree* is the one with the minimum height. We show by induction on h, the height of minimal derivation trees, that every tuple in $lfp(\mathcal{P}_1 \cup \ldots \cup \mathcal{P}_m)$ is obtained at some processor during the parallel evaluation, therefore is in the result of the

parallel evaluation.

If $h = 0$, by the definition of derivation trees, any tuple t with a minimal derivation tree of height 0 is in the EDB. Since the processing scheme is local, t must satisfy the potential usefulness wrt some subgoal of some rule in some $\mathcal{P}_i$, and be initially allocated at the processor where P_i is evaluated.

Assume that every tuple in $lfp(\mathcal{P}_1 \cup \ldots \cup \mathcal{P}_m)$ with a minimal derivation tree of height $h = 0, 1, \ldots, n$ is obtained at some processor during the parallel evaluation.

Let $h = n + 1$, t be a tuple in the least fixpoint with a minimal derivation tree of height h, and $r\theta$ be the successful rule instantiation formed by t and its children, where θ is some substitution. Suppose that r is in some restricted program $\mathcal{P}_i$ evaluated at processor s. By the definition of the successful rule instantiation and Definition 4.4, each ordinary subgoal $b\theta$ of $r\theta$ is in $lfp(\mathcal{P}_1 \cup \ldots \cup \mathcal{P}_m)$ and satisfies the potential usefulness wrt b of r. By the minimality, each $b\theta$ must have a minimal derivation tree of height n or less. By the induction hypothesis, $b\theta$ is obtained at some processor, say s', during the parallel evaluation. If $n + 1 = 1$, s and s' are the same processor. Otherwise, once obtained, $b\theta$ is sent from s' to s when line 2 or 9 of A1 is executed at s', received at s some time later when line 3, 10, or 17 of A1 is executed at s, and placed in the input relation and the incremental input relation of b of r when line 4, 11, or 18 of A1 is executed at s. In any case, when line 1 or 8 of A1 is executed in the next iteration, if all other ordinary subgoals of $r\theta$ are already received at s, t will be obtained at s. Otherwise, $b\theta$ will be in the input relation of b until the end of the parallel evaluation. Since the EDB is finite and rules are safe, minimal derivation trees are finite. Thus all ordinary subgoals of $r\theta$ will eventually arrive at s before the end of the evaluation, and t is obtained at s in the subsequent iteration.

A.3. Proof of Proposition 5.2

Proof: The first statement is true because that the outer loop in *Sending* iterates only on ΔP_i, $1 \le i \le K$, and that *Sending* follows immediately after *Initialization* and *Evaluation* in Algorithm A1. Within *Sending*, each tuple in ΔP_i is tested exactly once wrt to each ordinary subgoal that has the predicate p_i. Since in each iteration, ΔP_i in *Sending* contains tuples not yet added into P_i and these tuples are added into P_i in *Accumulation* at lines 5 and 12, no tuple appears in ΔP_i in different executions of *Sending*. Thus the second statement is also true.

A.4. Proof of Proposition 5.3

Proof: By Theorems 3.1 and 5.1, every successful instantiation of each rule in $\mathcal{P}$ is obtained at least once in the parallel evaluation.

Let r be a rule in $\mathcal{P}$ with a non–empty partition vector. By Definitions 3.2 and 3.3, for each substitution θ such that $r\theta$ is a successful instantiation of r, there is precisely one restricted rule r' of r such that $r'\theta$ is a successful instantiation of r'. Furthermore, θ is represented by the same set of ordinary subgoals in both rule instantiations. If $r'\theta$ is obtained no more than once, nor is $r\theta$. Since r' is evaluated at precisely one processor, we need to show only that $r'\theta$ is obtained no more than once at the processor. If r' is an exit rule, $r'\theta$ can only be obtained in line 1 of A1, which never repeats itself. So $r'\theta$ is obtained no more than once. If r' is not an exit rule and has more than one ordinary subgoal, $r'\theta$ may be obtained in line 8 of A1 at the processor where r' is evaluated or in line 9 of A1 at a processor from where a tuple satisfying the potential usefulness wrt a subgoal of r' is received. Since r' has more than one ordinary subgoal and the scheme is local, the test of data using potential usefulness does not need to obtain the successful rule instantiation. Since in different execution of line 8, whenever r' is evaluated by EVAL–INCR, at least one input relation is replaced by the corresponding incremental input relation which contains a set of tuples never used before, successful rule instantiations in different executions of line 8 can not possibly have the same body. As a result, $r'\theta$ is obtained no more than once.

ACKNOWLEDGMENT

The authors thank the anonymous referees for their invaluable suggestions and comments that have led to much improvement of the paper.

REFERENCES

[1] F.Bancilhon and R. Ramakrishnan, "An amateur's introduction to recursive query processing strategies," *Proc. ACM SIGMOD Conference*, pp. 16–52, 1986.

[2] J–P Cherney and C. Maindrevelle, "A parallel strategy for transitive closure using double hash–based clustering," *Proc. of 16th VLDB*, Brisbane, Australia, pp. 347–358, 1990.

[3] S. R. Cohen and O. Wolfson, "Why a single parallelization strategy is not enough in knowledge bases," *Proc. Symp. on PODS*, Philadelphia, Penn., pp. 200–216, March, 1989.

[4] S.K. Debray and N.W. Lin, "Static estimation of query sizes in Horn programs," *ICDT Conf.* Springer–Verlag Lecture Notes in Computer Science 470, pp. 514–528, 1990.

[5] G. Dong, "On distributed processibility of logic programs by decomposing databases," *Proc. ACM SIGMOD Conference*, Portland, OR, pp. 26–35, June, 1989.

[6] S. Ganguly, A. Silberschatz, and S. Tsur, "A framework for the parallel processing of datalog queries," *Proc. ACM SIGMOD Conference*, Atlantic City, NJ, pp.143–152, May, 1990.

[7] J.W. Lloyd, *Foundations of Logic Programming, Second, Extended Edition*, Springer–Verlag, 1987.

[8] J. Seib and G. Lausen, "Parallelizing Datalog programs by generalized pivoting," *Proc. Symp. on PODS*, Denver, Co., pp. 241–251, May, 1991.

[9] J. D. Ullman, *Principles of database and knowledge–base systems*, Vol. 1, Computer Science Press, Rockville, MD, 1988.

[10] J.L. Wolf, D.M. Dias, P.S. Yu, and J. Turek, "An effective algorithm for parallelizing hash joins in the presence of data skew," *IEEE Conf. on Data Engineering*, pp. 200–209, 1991.

[11] O. Wolfson, W. Zhang, H. Butani, A. Kawaguchi, and K. Mok, "A methodology for evaluating parallel graph algorithms and its application to single source reachability," Technical report, Dept. of EECS, UIC, 1992.

[12] O. Wolfson and A. Silberschatz, "Distributed processing of logic programs," *Proc. ACM SIGMOD Conference*, Chicago, IL., pp.329–336, 1988.

[13] O. Wolfson and A. Ozeri, "A new paradigm for parallel and distributed rule–processing," *Proc. ACM SIGMOD Conference*, Atlantic City, NJ, May, pp. 133–142, 1990.

[14] O. Wolfson, "Sharing the load of logic program evaluation," *Proc. Int'l. Symp. on Databases in Parallel and Distributed Systems*, Dec., 1988.

[15] W. Zhang, K. Wang, and S-C Chau, "Data partition and parallel evaluation of datalog programs," Technical report, Dept. of Math. and CS, Univ. of Lethbridge, 1992.

Weining Zhang received his B.Eng. degree in computer science and engineering in 1982 from the Chengdu Institute of Radio Engineering, Chengdu, Peoples Republic of China, and the M.S degree and Ph.D. degree in computer science in 1985 and 1988, respectively, from the University of Illinois, Chicago, USA.

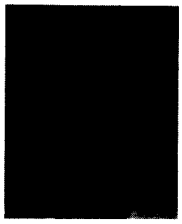
He was a researach scientist with the Centre for System Science, Simon Fraser University, British Columbia, Canada. Since 1989, he has been on the faculty in the Department of Mathematics and Comuter Science, University of Lethbridge, Alberta, Canada. His current research interests are in deductive database query optimization, distributed and parallel database query processing, and fuzzy database systems.

Dr. Zhang is a member of the IEEE Computer Society and the Association for Computing Machinery.

Siu-cheung Chau was born in Hong Kong. He received the B.Ed. degree in Mathematics fro the University of Lethbridge, Alberta, Canada in 1983, the M.Sc. and Ph.D. degrees in computing science from Simon Fraser University in 1984 and 1990, respectively. He joined the Mathemtics and computer Science department of the University of Lethbridge in 1984. He is currently an Associate Professor at the University of Lethbridge.

His research interests include fault-tolerant multicomputer systems, broadcasting algorithms, and datalog programs.

Dr. Chau is a member of the IEEE Computer Society and the ACM.



Ke Wang received his Ph.D. in 1986 from Georgia Institute of Technology, USA. His research interests include database theory, query processing, and deductive databases. Dr. Wang is currently a research fellow with the Department of Information Systems and Computer Science, National University of Singapore.