

Data Partition: A Practical Parallel Evaluation of Datalog Programs*

Weining Zhang, Ke Wang, Siu-Cheung Chau
Department of Mathematics and Computer Science
University of Lethbridge
Lethbridge, Alberta, Canada, T1K 3M4
e-mail: zhang@hg.uleth.ca

Abstract

Parallel processing provides an alternative to query optimization for evaluating logic programs in deductive databases. In this paper, previous parallel evaluation strategies based on the partition of rule instantiations are analyzed. We present a parallel evaluation strategy for general Datalog programs that is based on the partition of data and is more practical. A key issue is to determine a criterion of data transmission that reduces the amount of data transmitted and is tested efficiently. A notion of potential usefulness is given as such a criterion. The problem of designing appropriate partition schemes and processing schemes is addressed. Heuristics and algorithms are proposed for making decisions in the design process.

1 Introduction

Much work has been done on optimizing Datalog programs in deductive databases [1, 7]. As an alternative to query optimization, parallel evaluation strategies for Datalog programs are recently proposed in [2, 4, 5, 6, 10, 9, 11]. The idea behind these strategies is the program restriction [9], namely, appending conditions to rule bodies so that rule instantiations are partitioned among processors. These strategies fall into two categories.

Strategies in [2, 4, 6, 9, 11] are based on the syntactic characterization of decomposable programs which can be evaluated by a set of processors without communication or synchronization. These strategies apply only to restricted classes of programs and the recognition of decomposabilities is a difficult task.

Strategies in [5, 10] are for general Datalog programs. Each processor executes a modified semi-naïve algorithm and communicates with other processors. In each iteration, rules are evaluated using data either derived locally or received from other processors, in the previous iteration. After each iteration, newly derived data are sent to other processors according to conditions of partitioning rule instantiations, known as the restricting predicates. The evaluation terminates when no processor is busy and no data is in transmission.

*This work was supported in part by the Natural Sciences and Engineering Research Council of Canada.

A key issue in latter strategies is to determine the data transmission criterion. On one hand, to reduce the cost of communication and local evaluation, the criterion should effectively cut the amount of data transmission, while guaranteeing the correctness of the evaluation. On the other hand, the criterion should be tested efficiently. These requirements are difficult to meet simultaneously. The criterion must be carefully chosen, or the purpose of parallelization may be defeated by poor performances.

In [10], for a tuple to be sent from processor i to processor j , it must be in transmission set T_{ij} , and in turn, in set SR_j defined in [10] as follows.

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 .

where p_j is processor j and h_{gj} is the restricting predicate of rule r_g at processor j . When r_g is linear and all EDB relations are duplicated at processor i , condition K1 can be tested in time comparable to that of evaluating r_g at processor j , with h_{gj} appended to the body, using f and EDB . In general, the entire domain of constants have to be examined for instantiating r_g , or the decision of not sending a tuple based on the partially derived IDB may be incorrect. It is not clear how condition K1 can be tested efficiently.

The same problem exists in the strategy in [5] which is obtained through program rewriting. For every IDB predicate C , the set of tuples of C to be sent from processor i to processor j is defined by sending rules of the form:

$$C_{ij}(\vec{y}) : -C_{out}^i(\vec{y}), h(v(\tau)) = j,$$

where C_{out}^i is the set of all tuples of C derived at processor i , $C(\vec{y})$ is a subgoal in some rule τ , $h(v(\tau))$ is a discriminating function on a sequence of variables $v(\tau)$ in τ . To avoid repeating the evaluation work of the receiver (i.e., processor j), all variables in $v(\tau)$ are assumed to appear in at least one subgoal in τ [5]. But

because $C(\bar{y})$ may not necessarily be that subgoal if r is a nonlinear recursive rule, variables in $v(r)$ may not appear in $C_{out}^r(\bar{y})$, and the above sending rule may not be safe. Even if the rule is safe, it is still not clear how it can be efficiently evaluated without considering all possible instantiations of variables outside \bar{y} .

In this paper, we consider parallel evaluations of general Datalog programs and solve the above problems. The key is to partition data for individual subgoals rather than to partition rule instantiations in the first place. We make the following contributions. First, we generalize the definition of restricting predicate by allowing multiple partition functions in one rule. Secondly, a notion of potential usefulness defined based on local partition functions, i.e., each has its all variables in one subgoal, is identified as a criterion of data transmission that is tested efficiently. Finally, we consider design issues regarding partition functions and the assignment of computation load among processors. We identify a number of decision problems in the design process, and propose heuristics and algorithms for making these decisions. Results in this paper generalize previous approaches to parallel evaluation of programs.

The rest of the paper is organized as the following. Section 2 contains some preliminary definitions and notations. In Section 3, we present the data partition paradigm consisting of partition and processing schemes. In Section 4, we consider the design of partition scheme and processing scheme for a program. Section 5 concludes the paper.

2 Preliminary

We follow traditional definitions and assume readers are familiar with the terminology and concepts in deductive databases and logic programming [7].

In this paper, we only consider Datalog programs. A *positive literal* is either an atom of the form $q(\bar{x})$ where q is a predicate name and \bar{x} is a vector of variables or constants; or a built-in predicate, such as $x > 5$ or $u = y$. A *rule* is of the form $a : -b_1, b_2, \dots, b_k$ where a is an atom and each b_i is a positive literal. We call a the *head* and the conjunction of b_i 's the *body* of the rule. Each b_i is a *subgoal* of the rule. A subgoal is *ordinary* if it is an atom. We assume rules are *safe* [7]. A program is a finite set of rules. The semantics of a program is its least fixpoint which can be computed using the semi-naive bottom-up evaluation [1, 7].

To simplify the presentation, we define rectified rules. Rules for a derived predicate p are *rectified* if their heads are identical, and of the form $p(x_1, \dots, x_k)$ for distinct variables x_1, \dots, x_k , and in each rule, every argument of an ordinary subgoal is a variable appearing exactly once in ordinary subgoals. A program is rectified if rules for each derived predicate in the program are rectified. This definition differs from the one in [7] in that it requires not only the head but also all ordinary subgoals of rules to contain no constant nor repeated variable. Rules can be rectified by first using the method in [7] followed by considering, for each rule, argument positions of ordinary subgoals one by one from left to right. For each position, if it contains a constant a , then replace a with a distinct variable x

and append the subgoal $x = a$ to the rule body; if it contains a variable x that appears more than once in ordinary subgoals, then replace every occurrence of x in the rule body, except the first one, with a distinct variable y , and append the subgoal $x = y$ to the body. For example, consider the following rule.

$$p(x, y, y) : - e(x, 4, u, v), p(u, w, w), \\ p(y, v, m), f(m, v, u).$$

The rectified version of the rule is

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

3 Paradigm of Data Partition

In this section, we present the data partition paradigm for parallel evaluation of Datalog programs. Basically, for a given program, a set of restricted versions of the program, known as a processing scheme, is constructed based on generalized restricting predicates. Each restricted version is evaluated at exactly one processor using a parallel semi-naive algorithm. When evaluating a rule at a processor, only a subset of tuples, known as a fragment, is considered for instantiating each subgoal. Newly derived data are exchanged among processors, based on a criterion, through either a communication network or a shared memory. We consider three key issues of the paradigm, namely, the partition scheme and the processing scheme, the criterion for data transmission, and the parallel evaluation algorithm.

3.1 Partition Schemes and Processing Schemes

Our ideas are motivated by following examples.

Example 3.1 Consider the following program.

$$\rho_1 : q(x, y, z) : - q(u, x, v), q(y, v, w), q(w, z, t). \\ \rho_2 : q(x, y, z) : - e(x, y, z).$$

There are a number of ways to create restricted versions of the program. For simplicity, we only consider rule ρ_1 . A possible restricted version of rule ρ_1 for processor s_i is given by

$$\rho_3 : q(x, y, z) : -q(u, x, v), q(y, v, w), q(w, z, t), \\ (u + x + v) \bmod n = i.$$

where n is the number of available processors. Rule ρ_3 is responsible for instantiations of ρ_1 satisfying $(u + x + v) \bmod n = i$. Given a tuple $q(a, b, c)$, it may not be efficient to determine if the tuple will be used for a subgoal in some instantiation of ρ_3 because all instantiations of ρ_3 may need to be checked. We observe that $(u + x + v) \bmod n = i$ imposes a partition, $F_{<0>}, \dots, F_{<n-1>}$, on tuples of q , i.e., a tuple $q(a, b, c)$ is in $F_{<j>}$ iff $(a+b+c) \bmod n = j$. To find an instantiation of ρ_3 , only $F_{<i>}$ needs to be searched for tuples instantiating subgoal $q(u, x, v)$. But for instantiating the remaining two subgoals, it is not sufficient to consider only single fragment — all tuples of q must be considered. \square

Example 3.2 Suppose we rectify rule ρ_1 and have the following restricted version.

$$\begin{aligned}\rho_4 : q(x, y, z) : & -q(u, x, v), q(y, v_1, w), q(w_1, z, t), \\ & v = v_1, v^2 \bmod n_1 = i, v_1^2 \bmod n_1 = i, \\ & w = w_1, w \bmod n_2 = j, w_1 \bmod n_2 = j.\end{aligned}$$

where n_1 and n_2 are integers. The conditions specify three different partitions on tuples of q , one for each subgoal. The partition on $q(u, x, v)$ is specified by $v^2 \bmod n_1$, on $q(w_1, z, t)$ by $w_1 \bmod n_2$, and on $q(y, v_1, w)$ by both $v_1^2 \bmod n_1$ and $w \bmod n_2$. To find instantiations of ρ_4 , only their respective fragments need to be searched for tuples to instantiate the subgoals. There are $n_1 \times n_2$ versions of rule ρ_4 , corresponding to combinations of i, j , $0 \leq i \leq n_1 - 1$, $0 \leq j \leq n_2 - 1$. Notice that using rectified rules, we can specify different functions on originally same variables. This allows more flexibilities to partition relations of subgoals. \square

The key is to view the body of a rule as a join expression of relations and the body of a restricted version of the rule as the same join expression with some relations replaced by one of their fragments. To guarantee the correctness, versions of the rule wrt all combinations of fragments must be evaluated. We now formalize the ideas.

Definition 3.1 Let r be a rectified rule and $\vec{\alpha} = \langle x_1, \dots, x_m \rangle$ a vector of variables in r . A *partition function* (simply PF) $g(\vec{\alpha})$ over r is a mapping $g : D(x_1) \times \dots \times D(x_m) \rightarrow \mathcal{R}$, where $D(x_i)$ is the domain¹ of variable x_i ; and \mathcal{R} , called the *range* of g , is a subrange $[0..n-1]$ of integers for some $n \geq 1$.

Definition 3.2 Let r be a rectified rule and $G = \langle g_1(\vec{\alpha}_1), \dots, g_k(\vec{\alpha}_k) \rangle$, where $k \geq 0$, be a vector of partition functions over r with ranges $\mathcal{R}_1, \dots, \mathcal{R}_k$, respectively. The *domain* of G , $Dom(G)$, is the cross product $\mathcal{R}_1 \times \dots \times \mathcal{R}_k$. A *restricted rule* of r wrt G is obtained by appending to the body of r the conjunction $g_1(\vec{\alpha}_1) = v_1, \dots, g_k(\vec{\alpha}_k) = v_k$, where $V = \langle v_1, \dots, v_k \rangle \in Dom(G)$. The conjunction, denoted by $G = V$, is called the *constraint predicate* of r and each conjunct is a *partition constraint*. G , and V are called the *function vector* (FV) and the *processing vector* (PV), respectively. $Comp(r, G)$ denotes the set of all restricted rules of r wrt G , each corresponding to a PV $V \in Dom(G)$. \square

By definition, the restricted rule of any rule r wrt an empty FV $\langle \rangle$ is r itself. In this paper, when r, G and V are understood, we simply talk about a restricted rule without mentioning them.

¹Domains need not to be integers. Rules are always evaluated on domains required by the applications. Only when evaluating PFs, are values of variables mapped into integers. A simple such mapping is to use the internal representation of the values.

Definition 3.3 Let $\mathcal{P} = \{r_1, \dots, r_{\mathcal{M}}\}$ be a rectified program. A *partition scheme* \mathcal{G} of \mathcal{P} is a set $\{G_1, \dots, G_{\mathcal{M}}\}$, where each G_i is a (possibly empty) FV over r_i . A *restricted program* of \mathcal{P} wrt \mathcal{G} is a non-empty set containing, for every $1 \leq i \leq \mathcal{M}$, at most one restricted rule of r_i . A *processing scheme* of \mathcal{P} wrt \mathcal{G} is a set of restricted programs $\{\mathcal{P}_1, \dots, \mathcal{P}_m\}$ of \mathcal{P} wrt \mathcal{G} such that for $1 \leq i \leq \mathcal{M}$, every restricted rule of r_i wrt G_i is in exactly one \mathcal{P}_j . \square

Intuitively, the requirement of including at most one restricted rule of r_i in each restricted program is for simplicity. The requirement for each restricted rule to be included in exactly one restricted program is because that for the correctness of the evaluation, each restricted rule must be evaluated by at least one processor, and for efficiency, it should be evaluated by at most one processor.

Example 3.3 Consider the following program \mathcal{P} .

$$\begin{aligned}\mu_1 : q(x, y, z) : & -q(u, x, v), q(y, v_1, w), q(w_1, z, t), \\ & v = v_1, w = w_1.\end{aligned}$$

$$\mu_2 : q(x, y, z) : -e(x, y, z).$$

Suppose the partition scheme $\mathcal{G} = \{G_1, G_2\}$, where $G_1 = \langle f_1(v), f_2(v_1), f_3(w), f_4(w_1) \rangle$ and $G_2 = \langle \rangle$, and $f_1(v) = v^2 \bmod 2$, $f_2(v_1) = v_1^2 \bmod 2$, $f_3(w) = w \bmod 3$ and $f_4(w_1) = w_1 \bmod 3$. Then $Comp(\mu_1, G_1) = \{\mu_{i,j} \mid 0 \leq i \leq 1, 0 \leq j \leq 2\}$ where $\mu_{i,j}$ is given by

$$\begin{aligned}q(x, y, z) : & -q(u, x, v), q(y, v_1, w), q(w_1, z, t), \\ & v = v_1, w = w_1, f_1(v) = i, f_2(v_1) = i, \\ & f_3(w) = j, f_4(w_1) = j.\end{aligned}$$

and $Comp(\mu_2, G_2) = \{\mu_2\}$. A possible processing scheme of \mathcal{P} wrt \mathcal{G} is $\{\{\mu_{0,0}\}, \{\mu_{0,1}\}, \{\mu_{0,2}\}, \{\mu_{1,0}\}, \{\mu_{1,1}\}, \{\mu_{1,2}\}, \{\mu_2\}\}$. Another processing scheme of \mathcal{P} wrt \mathcal{G} is $\{\{\mu_{0,0}, \mu_2\}, \{\mu_{0,1}\}, \{\mu_{0,2}\}, \{\mu_{1,0}\}, \{\mu_{1,1}\}, \{\mu_{1,2}\}\}$ \square

Obviously, constraint predicates are generalization of restricting predicates [10] which only allow one function to be specified for a rule². Our definition allows multiple PFs in a constraint predicate. As we will see, this generalization is essential for efficient parallel evaluation.

3.2 Data Transmission Criteria

The major concern of communication is its cost, i.e., the amount of data to be transmitted. Central to the issue is a criterion for data to be sent among processors. For a parallel evaluation strategy to be practical, the criterion should allow both efficient test and effective reduction on the communication cost. To guarantee this, the test process must satisfy the following requirements.

²Although there is an example in [5] where the discriminating predicate contains a function composed by multiple functions, the concept and its importance on the efficient parallel evaluation is not addressed there.

1. For each tuple t and each processor s , the test whether t should be sent to s is done once for all, i.e., the same test should not be repeated despite that more data may be available later on.
2. The test depends solely on data available at the sending processor when the test is required. This is a basic requirement of a parallel, asynchronous evaluation.
3. The test does not affect the correctness of the evaluation, i.e., if a tuple is not sent to some processor because the criterion is unsatisfied, the result of the evaluation is the same as if the tuple is sent to the processor.

Not satisfying all these requirements may result in an incorrect or inefficient evaluation algorithm.

We now consider choices of the criterion. At one extreme, a criterion can be whether a tuple is newly derived. This forces every newly derived tuple to be broadcasted to all processors. Although the test is very efficient, the criterion does not cut communication cost in any way. At the other extreme, a criterion can be whether a tuple is useful at the receiving processor. This can be formally defined as the following.

Definition 3.4 Let \mathcal{P} be a rectified program, \mathcal{E} be an EDB, and r be a restricted rule of some rule in \mathcal{P} . t is *useful* to a subgoal b in r iff there exists an instantiation of r in which every ordinary subgoal is a tuple in the least fixpoint of \mathcal{P} wrt \mathcal{E} and t is the instantiated b . \square

Usefulness provides a lower bound of communication cost because every tuple transmitted is used by the receiving processor to derive some tuple. However, in many situations, this criterion can not be tested before the least fixpoint is completely evaluated since the test may violate either requirement 2 or 3. Moreover, since usefulness enforces relationships among subgoals via either shared variables or partition functions, in general, the test must consider all possible combinations of tuples for instantiating the rule. As a result, the test has a time complexity comparable to that of evaluating the rule at the receiving processor. (Although some syntactic properties of a program, e.g., pivoting property [6], may reduce the test to a trivial case, general programs do not have such properties.) Based on these observations, we consider a weaker criterion that deals with local PFs only.

Definition 3.5 Let $f(\vec{x})$ be a partition function over a rectified rule r . $f(\vec{x})$ is *local* to an ordinary subgoal q in r if all variables in \vec{x} appear in q ; otherwise, $f(\vec{x})$ is *global*. The function vector G over r is *local* if every function in G is local to some ordinary subgoal in r . A partition scheme $\mathcal{G} = \{G_1, \dots, G_m\}$ is *local* if every function vector G_i is local. A processing scheme is *local* if it is based on a local partition scheme. \square

Definition 3.6 Let r be a restricted rule of a rule with a local FV and let b be an ordinary subgoal in r . A tuple t is *potentially useful* to b in r if t is a tuple of

Algorithm A1:
Initialization;
Sending;
Receiving;
Decomposition;
REPEAT
 REPEAT
 Evaluation;
 Sending;
 Receiving;
 Decomposition
 UNTIL $\Delta Q' = \emptyset$ for all q ;
 IF NOT Termination THEN BEGIN
 Wait for data from other processors;
 Receiving;
 Decomposition
 END
UNTIL Termination;

Figure 1: Semi-naive Evaluation at Site s

the predicate of b and all partition constraints³ local to b are evaluated true on t . \square

Based on this criterion, tuples are filtered for transmission by conditions introduced by the constraint predicates local to the relevant subgoals. Clearly, transmitted tuples are not guaranteed to be successfully used to derive tuples at the receiving processor. On the other hand, the test of the criterion is very efficient as indicated by the following proposition.

Proposition 3.1 Whether a tuple is potentially useful to a subgoal b in a restricted rule r can be tested in the time required to evaluate all partition functions local to b on the tuple. \square

Here, we ignore the time of evaluating instantiated built-in predicates, such as $1 \leq 2$. If all local partition functions are evaluated in a constant time, which is a reasonable assumption, potential usefulness can be tested in a constant time. By being restricted to local processing schemes, our strategy is more practical than previous ones. For the rest of paper, we consider only local processing schemes.

3.3 An Evaluation Algorithm

We now present an algorithm for evaluating a restricted program. The algorithm is a modification of the parallel semi-naive algorithm in [10]. We assume each restricted program is evaluated at exactly one processor and EDB are initially distributed based on potential usefulness criterion. In the following, q is a derived predicate, Q is the partial relation for q stored at the processor, ΔQ is the set of new tuples of q locally derived in the last iteration, $\Delta Q'$ is the set of tuples of q received through communication in the last iteration with tuples in Q removed. Also, we denote

³To achieve better efficiency, we can also include all built-in subgoals of r that have all their variables in b here.

by P_b the set of tuples to be substituted for subgoal b of predicate p in evaluation. When all processors terminate, the least fixpoint of the program is the union of all tuples in Q stored at all processors, for all derived predicates q . The algorithm is given in figure 1 and consists of following procedures.

Initialization: for every derived predicate q , compute the initial set of tuples of q by evaluating exit rules of q using EDB locally stored. This can be done by computing an expression similar to EVAL in [7].

Evaluation: for each derived predicate q , compute new tuples ΔQ of q by evaluating every recursive rule of q with each subgoal b of a predicate p substituted by P_b . This can be done by computing an expression similar to EVAL-INCR in [7].

Sending: send to every processor the tuples in ΔQ that are potentially useful to some subgoal owned by that processor. Tuples for different subgoals of the same predicate may be packed together before transmitted to avoid duplicate transmissions.

Receiving: for each derived predicate q , add to $\Delta Q'$ all tuples of q received in the last iteration, remove from $\Delta Q'$ tuples already in Q , and add all tuples in $\Delta Q'$ to Q .

Decomposition: for every subgoal b of a derived predicate q , add to Q_b the tuples in $\Delta Q'$ that are potentially useful to the subgoal. This procedure assigns the received tuples to subgoals according to potential usefulness.

Termination: Same as in [5, 10].

Theorem 3.2 Let \mathcal{P} be a rectified program and $\{\mathcal{P}_1, \dots, \mathcal{P}_m\}$ be a local processing scheme of \mathcal{P} . If each \mathcal{P}_i is evaluated at a processor s_i using algorithm A1, the set of all processors $\{s_1, \dots, s_m\}$ correctly computes the least fixpoint of \mathcal{P} . \square

4 Design of Schemes

Given a rectified program $\mathcal{P} = \{r_1, \dots, r_K\}$ and \mathcal{N} processors, We want to design a partition scheme $\mathcal{G} = \{G_1, \dots, G_K\}$ and a processing scheme $\{\mathcal{P}_1, \dots, \mathcal{P}_m\}$ wrt \mathcal{G} so that each \mathcal{P}_i is evaluated at exactly one processor. Since each restricted program is evaluated by exactly one processor, \mathcal{G} must be chosen so that $m \leq \mathcal{N}$. On the other hand, to enhance parallelism, \mathcal{G} should be designed to allow m to be as close to \mathcal{N} as possible. More importantly, we want the partition scheme to meet the demands for relations of certain subgoals to be partitioned. In general, partitioning large relations is more desirable than partitioning small ones. These optimization requirements make the design non-trivial.

We identify the following steps of the design process.

1. *Design formats of the partition scheme.* For each rule, decide which subgoals need to be partitioned, how many partition functions are for each

subgoal, which variables a partition function has as arguments, what operation each function performs, how functions are related, etc.

2. *Determine ranges of partition functions.*

3. *Design the processing scheme.* Group restricted rules to create restricted programs.

We separate function ranges from their format because ranges are dependent on the number of available processors, while format of functions is not. In order to obtain a good design, decisions made in each step may have to be readjusted several times before they are finalized. In fact, readjusting decisions is a means to fine-tune the resulting schemes for certain applications. We now give an example to illustrate the steps involved in the design process.

Example 4.1 Consider the following rectified program

$$\begin{aligned} \sigma_1 : q(x, y, z) : & -b(y, v_1, w_1), q(u_1, x, v_2), \\ & q(u_2, u_3, w_2), t(z, v_3), v_1 = v_2, \\ & w_1 = w_2, u_1 = u_2, v_2 = v_3, u_2 = u_3. \end{aligned}$$

$$\begin{aligned} \sigma_2 : q(x, y, z) : & -c(w_3, v_4, z), q(u_4, v_5, y), \\ & d(z, w_4, u_5), w_3 = w_4, v_4 = v_5, u_4 = u_5. \end{aligned}$$

$$\sigma_3 : q(x, y, z) : -e(x, y, z).$$

Assume $\mathcal{N} = 6$. We now make decisions at each step without giving explanations (justification will be given in subsequent subsections). First, we choose the following format of partition scheme \mathcal{G} . $f_1(u_1) = u_1 \bmod n_1$ on $q(u_1, x, v_2)$ in σ_1 ; $f_2(v_2) = v_2^2 \bmod n_2$ on $q(u_1, x, v_2)$ in σ_1 ; $f_3(u_2) = u_2 \bmod n_1$ on $q(u_2, u_3, w_2)$ in σ_1 ; $f_4(w_3, v_4, z) = (w_3^2 + v_4^2 + z^2) \bmod n_4$ on $c(w_3, v_4, z)$ in σ_2 ; $f_5(u_4) = u_4 \bmod n_5$ on $q(u_4, v_5, y)$ in σ_2 ; $f_6(x, y, z) = (x + y + z) \bmod n_6$ on $e(x, y, z)$ in σ_3 ; where n_i 's are the sizes of ranges of functions to be determined in the next step. Notice that we have chosen f_1 and f_3 over σ_1 to be the same mapping (on different variables). $\mathcal{G} = \{G_1, G_2, G_3\}$, where the function vectors $G_1 = \langle f_1, f_2, f_3 \rangle$, $G_2 = \langle f_4, f_5 \rangle$, $G_3 = \langle f_6 \rangle$. Next, we choose the ranges to be $n_1 = 3$, $n_2 = 2$, $n_4 = 2$, $n_5 = 3$, $n_6 = 6$. Since f_1 and f_3 are defined on variables u_1 and u_2 such that $u_1 = u_2$, the following tables give all processing vectors of rules in question.

PVT T_{σ_1}			PVT T_{σ_2}		PVT T_{σ_3}
f_1	f_2	f_3	f_4	f_5	f_6
0	0	0	0	0	0
0	1	0	1	0	1
1	0	1	0	1	2
1	1	1	1	1	3
2	0	2	0	2	4
2	1	2	1	2	5

Finally, a processing scheme can be formed by grouping row i of the three tables to create a restricted

program \mathcal{P}_i . (There are other ways to group these processing vectors). For instance, \mathcal{P}_3 is given by

$$\begin{aligned} q(x, y, z) : & -b(y, v_1, w_1), q(u_1, x, v_2), q(u_2, u_3, w_2), \\ & t(z, v_3), v_1 = v_2, w_1 = w_2, u_1 = u_2, v_2 = v_3, \\ & u_2 = u_3, f_1(u_1) = 1, f_2(v_2) = 0, f_3(u_2) = 1. \end{aligned}$$

$$\begin{aligned} q(x, y, z) : & -c(w_3, v_4, z), q(u_4, v_5, y), \\ & d(z, w_4, u_5), w_3 = w_4, v_4 = v_5, u_4 = u_5, \\ & f_4(w_3, v_4, z) = 0, f_5(u_4) = 1. \end{aligned}$$

$$q(x, y, z) : -e(x, y, z), f_6(x, y, z) = 2. \square$$

4.1 Format of Partition Scheme

In this subsection, we consider decisions to be made in order to obtain the format of a partition scheme, and discuss heuristics to be used in making these decisions.

Choosing Subgoals for Partition: A number of factors may affect the decision. One factor is the size of relations of subgoals. A heuristic is to partition subgoals with large relations. One may apply methods known in the literature [3] or use historical statistical data from previous execution of the programs to estimate the size of relations. A second factor is the syntax of the program. For instance, if a program is pivoting, one can choose the subgoals suggested by the pivoting property to eliminate the communication cost. Another factor is the relationships among subgoals with the same predicate name. One may insist that all subgoals (in the same rule) with the same predicate name are either all chosen or all not chosen. Similar heuristic applies to subgoals with shared variables (in the original, un-rectified program, of course). Due to the restriction on the number of available processors, it is possible that all partition functions of a chosen subgoal are void in the sense that their ranges have size one, so that the subgoal is not partitioned at all. This problem can be solved by assigning a priority value to each chosen subgoal and keep PFs of high priority subgoals from being void.

Number Of Partition Functions: For each chosen subgoal, one or more partition function can be specified. There are two reasons to use multiple partition functions for a subgoal. First, a subgoal may involve more than one join with other subgoals, one function may be defined on each *join variable*, i.e., variables between which equalities can be logically inferred from the equalities in the rule. Second, multiple partition functions can impose a "composite hash partition" which is an important tool for handling data skew [8]. But, increasing the number of partition functions carelessly may result in many void partition functions. Thus it may need to go back and forth between steps that determine the number of partition functions and the ranges of partition functions until a satisfactory decision is made.

Arguments and Operations of Partition Functions: Arguments and operations of a partition function are determined based on the intended use of the partition function and balanced size of fragments. For example,

if a join operation is to be partitioned, partition functions defining the same mapping on the join variables should be specified. This is illustrated by functions f_1 and f_3 in Example 4.1. The choice of operations will affect sizes of fragments. Presumably, all fragments should have nearly the same size.

Dependencies of Partition Functions: In Example 4.1, the values of partition functions f_1 and f_3 are always equal in every processing vector of rule σ_1 . This type of dependency is due to the fact that variables u_1 and u_2 in f_1 and f_3 , respectively, are join variables. If this dependency is enforced, then only 6 out of 18 PVs are needed to generate restricted rules of σ_1 . There is another type of dependency in Example 4.1. Functions f_3 and f_5 define the same partition on the relation of q , i.e., they hash on the first argument of the respective subgoals using the same operation and range. If this dependency is enforced, by appropriately grouping versions of rules, the relevant subgoals can share the same fragment of q . This can reduce communication cost. These dependencies are formally defined as follows.

Definition 4.1 Let b and b' be ordinary subgoals in a rectified program. Let $f(x_1, \dots, x_k)$ and $g(y_1, \dots, y_k)$ be partition functions local to b and b' , respectively. $f(x_1, \dots, x_k)$ and $g(y_1, \dots, y_k)$ are *equivalent* if for every constant vector $\langle a_1, \dots, a_k \rangle$, $f(a_1, \dots, a_k) = g(a_1, \dots, a_k)$. $f(x_1, \dots, x_k)$ and $g(y_1, \dots, y_k)$ are *join related* if 1) they are equivalent; 2) b and b' are different subgoals in the same rule; 3) for $1 \leq i \leq k$, x_i and y_i are join variables. $f(x_1, \dots, x_k)$ and $g(y_1, \dots, y_k)$ are *synchronous* if 1) they are equivalent; 2) b and b' are the same subgoal or they are in different rules with the same predicate name; 3) x_i and y_i appear at the same argument position in b and b' , respectively. \square

Each of the two dependencies imposes an equivalence relation on PFs. The corresponding equivalence class are called the *join classes* and *synchronous classes*, respectively. In Example 4.1, functions $f_1(u_1)$ and $f_3(u_2)$ are join related; $f_3(u_2)$ and $f_5(u_4)$ are synchronous. Join classes can be used to determine ranges of PFs and to eliminate useless processing vectors; synchronous classes can be used to optimize formation of restricted programs.

4.2 Ranges of Partition Functions

Ranges of partition functions are determined one rule at a time. Let $G = \langle f_1, \dots, f_k \rangle$ be a local PV over a rule τ and \mathcal{N} be the total number of processors. The problem is to determine the ranges of f_i , $1 \leq i \leq k$, subject to constraints imposed by \mathcal{N} (to be described later). Since the range of f_i , $\mathcal{R}_i = [0..n_i - 1]$, is uniquely determined by the size n_i , in the sequel, we refer n_i as the range of f_i .

Recall that each restricted rule of τ is determined by a processing vector $V = \langle v_1, \dots, v_k \rangle$, where $v_i \in \mathcal{R}_i$. Thus the total number of versions of τ is given by $n_1 \times \dots \times n_k$. Since each processor evaluates no more than one restricted rule of τ , the total number of processing vectors must be no more than the number of processors. In addition, it is also desirable for ranges to satisfy the following requirements.

1. The number of processing vectors is as close to the total number of processors as possible, so that the maximum utilization of processors is achieved.
2. The number of void partition functions is as fewer as possible, so that the maximum utilization of partition functions is achieved.

When join related dependencies are enforced, in each processing vector, the values of functions in the same join class must be equal. Therefore as far as ranges are concerned, it is the join classes rather than individual functions that need to be considered.

Let m be the number of join classes of f_1, \dots, f_k . Let x_i , $1 \leq i \leq m$, be variables representing the m ranges to be found for the join classes. The problem is to find an assignment of integer values to x_i , $1 \leq i \leq m$, such that, the following conditions are satisfied, if possible.

1. $x_i \geq 1$;
2. $\prod_i x_i \leq \mathcal{N}$;
3. $\mathcal{N} - \prod_i x_i$ is minimized;
4. the number of x_i that is less than 2 is minimized.

Note that any three conditions including Conditions 1 and 2 can always be satisfied at the same time. If not all four conditions are satisfiable, one of conditions 3 and 4 must be compromised. The following example illustrates a situation where this happens.

Example 4.2 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_i s 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 there are two rather than one x variables having 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; 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. \square

In the sequel, we present two practical methods to find values of x_i s, one emphasizing on condition 4, the other on condition 3. The usefulness of these methods depends on applications.

The first method is given in Figure 2. The idea is to assign ranges evenly. For any $\mathcal{N} \geq 1$, there exists some integer $d \geq 1$, s.t., $d^m \leq \mathcal{N} \leq (d+1)^m$. That is, the value of each x_i is either d or $d+1$. We first assign to each x_i the value d . Then, one x at a time, we increase the value of the variable by 1, until the product of x_i s exceeds \mathcal{N} . The algorithm is very efficient. Applying algorithm A2 to example 4.2, we can obtain the first assignment that satisfying conditions 1, 2 and 4. It can be shown that for small m or large \mathcal{N} , algorithm A2 returns ranges that make good use of processors.

Now we describe the second method which satisfies Conditions 1, 2, and 3 by assigning values to x_i s so that $\prod_i x_i = \mathcal{N}$. The idea is to factorize \mathcal{N} into prime numbers greater than 1, and assign these numbers to x_i s. There are two cases.

Algorithm A2

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

Output: Ranges of join classes satisfying conditions 1, 2 and 4.

Method:

```

 $d := \lfloor 2^{(\log_2 \mathcal{N})/m} \rfloor$ ;
FOR  $i = 1$  TO  $m$  DO  $x_i := d$ ;
 $K := d^m$ ;
 $C := (d + 1)/d$ ;
FOR  $i := 1$  TO  $m$  DO
  IF  $K \times C \leq \mathcal{N}$  THEN
     $K := K \times C$ ;
     $x_i := x_i + 1$ 
  ELSE RETURN( $x_1, \dots, x_m$ );

```

Figure 2: Algorithm for even assignment of ranges

Case 1: The number of prime factors of \mathcal{N} is less than m . In this case, x_1 is assigned the largest factor, x_2 the second largest factor, etc. When all factors are assigned, the remaining x_i s are simply assigned the value 1.

Case 2: The number of prime factors is greater than m . In this case, each x_i is assigned one or more prime factor so that the value of x_i is the product of the prime factors assigned to it.

A very simple method for finding the factors is to use \mathcal{N} as the initial quotient, and repeatedly find the smallest prime number other than 1 that evenly divides the previous quotient. This process terminates when the quotient becomes one. The second assignment in Example 4.2 is obtained by applying this method.

Once values for x_i s are obtained, they are assigned to join classes. The simplest method is to arbitrarily assign one x_i to exactly one join class, so that every function in the class has a range $[0..x_i - 1]$. A better method is to define priority values of join classes based on priority values of subgoals, and assign larger x_i to join class with higher priority.

Partition functions are completely specified once their format and ranges are specified. The partition scheme is then given by the set of function vectors of rules, each being a (possibly empty) sequence of non-void partition functions over a rule.

4.3 Grouping Processing Vectors

Let \mathcal{P}' be the set of rules with non-empty function vectors in a rectified program \mathcal{P} and $G = \langle f_1, \dots, f_m \rangle$ be the function vector of a rule r in \mathcal{P}' , with ranges n_1, \dots, n_m , where $n_i > 1$, $1 \leq i \leq m$. We create a table T_r , called the *processing vector table* (or simply PVT) of r (wrt G), that contains all processing vectors $\langle v_1, \dots, v_m \rangle$ with $0 \leq v_i \leq n_i - 1$, such that $v_i = v_j$ if f_i and f_j are in the same join class. Assume PVTs are obtained for every rule in \mathcal{P}' . Since ranges

of partition functions are decided individually for each rule, the sizes of PVTs are not necessarily equal.

To construct restricted programs of \mathcal{P} , we only need to group the corresponding PVs. A simple way of doing so is to select an arbitrary PV from every non-empty PVT to form a group, and then to delete the selected PVs from their PVTs. This repeats until all PVTs are empty. PVs in the same group are used to create restricted rules in the same restricted program. Rules with empty PVs are either included into $\mathcal{P}_1, \dots, \mathcal{P}_j$ 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.

In general, optimization is possible in this step. In Example 4.1, $f_3(u_2)$ and $f_5(u_4)$ are synchronous and therefore 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 this restricted program use different fragments of q . Thus the processor evaluating this version must store data from two fragments of q . However, if this synchronous dependency needs to be enforced, we must switch rows 2 and 3 in T_{σ_2} . Then the two subgoals will share the same fragment, which reduces the communication cost. The idea is to reduce the number of distinct fragments stored at a processor. Due to space limit, we will not discuss it further.

5 Conclusion

In this paper, we present a practical parallel evaluation strategy for general Datalog programs based on the data partition. A key issue is to determine a criterion for data transmission that reduces the amount of data transmitted and is efficiently testable. Potential usefulness is given as such a criterion. The problem of designing partition schemes and processing schemes for a given program is addressed. Heuristics and algorithms are provided for making decisions in the design process.

There are some immediate extensions of the results in this paper. Similar to [10], the strategy can be easily extended to stratified Datalog programs with negations. Also, definitions of constraint predicates can be extended to allow conditions like $2 \leq u \ \& \ u \leq 5$ to be specified. Such conditions can be used to generate even size fragments and specify relevant fragments of subgoals for joins.

A challenging open problem is to define appropriate measurements for the performance of parallel evaluation strategies. In this regard, very little result is known. Another open problem is to find an efficient algorithm to group processing vectors such that communication cost and storage requirement are minimized.

References

- [1] F. Bancilhon and R. Ramakrishnan, "An amateur's introduction to recursive query processing strategies", *Proc. ACM SIGMOD Conference*, 1986, pp. 16-52.
- [2] S. R. Cohen and O. Wolfson, "Why a single parallelization strategy is not enough in knowledge bases", *Proc. Symp. on PODS*, Philadelphia, Penn., March, 1989, pp. 200-216.
- [3] 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, 1990, pp. 514-528.
- [4] G. Dong, "On distributed processibility of logic programs by decomposing databases", *Proc. ACM SIGMOD Conference*, Portland, OG, June, 1989, pp. 26-35.
- [5] S. Ganguly, A. Silberschatz and S. Tsur, "A framework for the parallel processing of datalog queries", *Proc. ACM SIGMOD Conference*, Atlantic City, NJ, May, 1990, pp.143-152.
- [6] J. Seib and G. Lausen, "Parallelizing Datalog programs by generalized pivoting", *Proc. Symp. on PODS*, Denver, Co., May, 1991, pp. 241-251.
- [7] J. D. Ullman, *Principles of database and knowledge-base systems*, Vol. 1, Computer Science Press, Rockville, MD, 1988.
- [8] 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*, 1991, pp. 200-209.
- [9] O. Wolfson and A. Silberschatz, "Distributed processing of logic programs", *Proc. ACM SIGMOD Conference*, Chicago, IL., 1988, pp.329-336.
- [10] O. Wolfson and A. Ozeri, "A new paradigm for parallel and distributed rule-processing", *Proc. ACM SIGMOD Conference*, Atlantic City, NJ, May, 1990, pp. 133-142.
- [11] O. Wolfson, "Sharing the load of logic program evaluation", *Proc. Int'l. Symp. on Databases in Parallel and Distributed Systems*, Dec., 1988. pp. 46-55.