On Distributed Processibility of Datalog Queries by Decomposing Databases*

Guozhu Dong
Computer Science Dept
University of Southern California
Los Angeles, CA 90089-0782
dong@cse.usc.edu

Abstract We consider distributed or parallel processing of datalog queries. We address this issue by decomposing databases into a number of subdatabases such that the computation of a program on a database can be achieved by *unioning* its *independent evaluations* on the subdatabases. More specifically, we identify two kinds of distributed-processible programs according to the properties of database decomposition. (i) A program is *disjoint distributive* if it is distributed processible over a decomposition consisting of subdatabases with disjoint domains. A characterisation of such programs is given in terms of an easily decidable syntactic property called *connectivity*. (ii) A program is *bounded distributive* if it is distributed processible over a decomposition consisting of subdatabases with a fixed size. Three interesting characterisations of such a program are presented, the first by bounded recursion, the second by equivalence to a 1-bounded-recursive program, and the third by constant parallel complexity

1 Introduction

We introduce a new approach of computing datalog programs in parallel. This approach is suitable for an arbitrary number of processors, and does not require inter-processor communication. Intuitively, parallelism is obtained by decomposing databases into a number of subdatabases such that the computation of a program on a database can be achieved by *unioning* its *independent evaluations* on the subdatabases. As shall be seen, this approach can lead to a significant speed-up in datalog program evaluation in many situations.

*This work was supported in part by the NSF grant CCR-86-18907. A portion of this work is drawn from the author's Ph.D. thesis written under the supervision of Professor S. Ginsburg.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1989 ACM 0-89791-317-5/89/0005/0026 $1.50

Let us demonstrate the approach using two examples.

Example 1 Consider the transitive closure program. The arcs of a graph are represented by facts in a relation A, and the arcs in the transitive closure are given by facts in a relation T. The program is written in DATALOG (a dialect of PROLOG).

$$T(x,z)\&A(z,y) \rightarrow T(x,y),$$

$$A(x,y) \rightarrow T(x,y).$$

If the relation A is partitioned into connected components of the graph, then T can be calculated by unioning the evaluation of the transitive closure of these connected components. In other words, the transitive closure is distributed processible by decomposing the database into subdatabases with disjoint sets of constants. Note that no facts are computed by more than one processor. □

Example 2 Consider the following bounded-recursive program.

$$likes(x,y) \rightarrow buys(x,y),$$

$$trendy(x)\&buys(z,y) \rightarrow buys(x,y).$$

It can be verified that the computation of the program on each database can be achieved by unioning the evaluation of the program on all subdatabases containing two facts from the database. In other words, the computation is distributed processible by decomposing the database into subdatabases with a fixed size 2. Note that some fact may be computed by two or more processors. □

The distribution of computation load by decomposing databases as considered in this paper has several nice properties. (i) The result is always the same as computing the program on the whole database with a single processor. (ii) The computations on different processors are independent, i.e., no communication is necessary. (iii) There is no limitation on the number of processors. Indeed, efficiency can be achieved with an arbitrary number of processors. (iv) There is no need to modify the program, i.e., the same program is evaluated on all the processors.

In addition to giving results on distributed processing by decomposing databases, our approach provides a classification scheme for datalog programs. In particular, we identify two kinds of distributed-processible programs according to the properties of database decomposition. (i) A program is *disjoint distributive* if it is distributed processible over a decomposition consisting of subdatabases with disjoint constants. (Example 1.) A characterization of such programs is given in terms of an easily decidable syntactic property called *connectivity.* (ii) A program is *bounded distributive* if it is distributed processible over a decomposition consisting of subdatabases with a fixed size. (Example 2.) Three interesting characterizations of such a program are presented, the first by bounded recursion, the second by equivalence to a 1-bounded-recursive program, and the third by constant parallel complexity.

In general, this work is concerned with the efficient evaluation of datalog programs (or queries), which has recently become a very important and active area of research. (See Proceedings of PODS 85-88, Proceedings of SIGMOD 86-88, Proceedings of VLDB 85-88, and Proceedings of Expert Database Systems 88.) In particular, this study is closely related to two major aspects of the current interest. The first aspect is on the parallel evaluation of datalog programs. In [CoKa, UlVG] the parallel complexity issue was considered. In [WoSi, Wolf] an actual approach, namely predicate decomposability, was used to distribute work load among processors and a number of characterizations on single rule programs with respect to such decomposability were presented. In a following paper [CoWo], a number of parallelization strategies (classified in terms of evaluaton cost, communication overhead, and synchronization among processors) for logic programs were proposed and it was argued that the appropriate strategy depends on the program being evaluated. In contrast, database decomposability is utilized in this paper for load distribution or parallel processing of datalog programs. The second aspect is on the identification of nice classes of programs, such as linear recursion [BaRa] and bounded recursion [Ioan, Naug, NaSa, Vard], for the existence of special methods of efficient evaluation. In [Dong] programs sequentially decomposable into single-rule programs were considered. The decomposability of databases provides a new scheme for identifying nice classes of datalog programs. In addition, this work provides a new angle to understanding bounded recursiveness.

The rest of this paper is organized into five sections. The first, Section 2, provides the necessary preliminaries. Section 3 formalizes the concept of decomposition of databases for parallel processing. Section 4 examines the disjoint-distributive programs. Section 5 considers the bounded-distributive programs. Section 6 presents our conclusion and some further questions.

2 Preliminaries

In this section we review the formalisms of relational databases, datalog programs and datalog mappings.

We start with some fundamentals of relational databases.

We assume throughout three pairwise disjoint infinite sets, Dom_∞, N, and $\mathcal{V}$, of abstract elements, called <u>constants</u> (or <u>domain values</u>), <u>relation names</u>, and <u>variables</u> respectively. We shall usually use integers to denote the elements of Dom_∞, R (possibly with a subscript) the elements of N, and v, x, y, and z (possibly with a subscript) the elements of $\mathcal{V}$. We also assume that α is a total function from N to the set of positive integers. For each R in N, α(R) is the <u>arity</u> of R.

The above assumptions are concerned with the most basic elements. We now use them to introduce some general relational database terminology. In particular, a <u>fact</u> is an expression of the form

$R(a_1, \cdots, a_{\alpha(R)})$, where R is a relation name and $a_1, \cdots, a_{\alpha(R)}$ are constants. A database scheme is a nonempty finite subset of N. For each database scheme $\mathbf{D}$, a fact $R(a_1, \cdots, a_k)$ is said to be over $\mathbf{D}$ if R is in $\mathbf{D}$; and a database over $\mathbf{D}$ is a finite set of facts over $\mathbf{D}$. For each database D, let $dom(D) = \{a$ in $Dom_\infty \mid a$ occurs in $D\}$.

For example, let $\mathbf{D} = \{Ancestor, R\}$, where *Ancestor* and R are relation names with arity 2. Then $\mathbf{D}$ is a database scheme and $\{R(1,2), R(2,3), Ancestor(1,3)\}$ is a database over $\mathbf{D}$.

We next present the notions of Horn clauses, datalog programs, and datalog mappings. Such programs and mappings have been used widely in database systems as queries.

The usual definition in the literature of a Horn clause [Ll] is very general. Specifically, constants, equality and negation symbols are allowed and clauses without a "head" or "body" are permitted. For our purpose, the simplest definition is adopted here. Formally, a formula is an expression of the form $R(v_1, \cdots, v_{\alpha(R)})$, where R is a relation name and $v_1, \cdots, v_{\alpha(R)}$ are variables. A Horn clause is an expression of the form

$$A_1\&...\&A_m \rightarrow H,$$

where $m \geq 1$, $A_1, \cdots, A_m$ and H are formulas, and each variable appearing in H appears in some A_i. A Horn clause is over a database scheme $\mathbf{D}$ if each relation name occurring in the Horn clause occurs in $\mathbf{D}$. For each Horn clause $r = A_1\&...\&A_m \rightarrow H$, $A_1\&...\&A_m$ and H are called the body and the head, respectively, of r, and denoted $B(r)$ and $H(r)$, respectively.

We sometimes write $B(r)$ as $\{A_1, \cdots, A_m\}$ and r as $B(r) \rightarrow H(r)$. As is usually done in the literature, we shall also use the term "rule" as a synonym for "Horn clause."

To illustrate the above concepts, let *Ancestor* and R be relation names, each with arity 2. Then "$R(x,y)$" is a formula whereas "$R(1,z)$" is not. The expressions "$Ancestor(x,y)\&Ancestor(y,z) \rightarrow Ancestor(x,z)$" and "$R(x,y) \rightarrow R(y,x)$" are Horn clauses. However, the expressions "$R(x,y) \rightarrow R(x)$," "$\rightarrow R(x,y)$," "$R(x,x) \rightarrow R(1,1)$," and "$R(x,y) \rightarrow R(x,z)$" are not Horn clauses.

We are now ready to define datalog programs, these being a dialect of general logic programs.

Definition. A datalog program (over a database scheme $\mathbf{D}$) is a finite set of Horn clauses (over $\mathbf{D}$).

Clearly, if P is a datalog program over $\mathbf{D}$ and $\mathbf{D}'$ is a database scheme containing $\mathbf{D}$, then P is a datalog program over $\mathbf{D}'$.

Note that our version of datalog program allows only safe Horn clauses using relation names and variables. It is easy to see that this work can be extended to the general case (with possibly modified results).

For each datalog program, one can associate a mapping on all the databases of interest. First though, some preliminary notions are needed before the definition of such a mapping can be given.

An assignment is a mapping from $\mathcal{V}$ into Dom_∞. For each assignment τ and formula $R(x_1, \cdots, x_k)$, let $\tau(R(x_1, \cdots, x_k))$ denote the fact $R(\tau(x_1), \cdots, \tau(x_k))$. For each database scheme $\mathbf{D}$, let $\mathcal{S}_\mathbf{D}$ denote the set of all databases over $\mathbf{D}$.

We are now able to define datalog mappings.

Definition. Suppose $\mathbf{D}$ is a database scheme and P is a datalog program over $\mathbf{D}$. Let P^1 be the mapping from $\mathcal{S}_\mathbf{D}$ to $\mathcal{S}_\mathbf{D}$ defined (for each D in $\mathcal{S}_\mathbf{D}$) by

$$P^1(D) = D \cup D',$$

where $D' = \{\phi(F) \mid$ there exist $A_1\&\cdots\&A_k \rightarrow F$ in P and assignment ϕ such that $\phi(A_i)$ is in D for each i, $1 \leq i \leq k\}$. For each $i \geq 1$, let P^{i+1} be the mapping from $\mathcal{S}_\mathbf{D}$ to $\mathcal{S}_\mathbf{D}$ defined (for each D in $\mathcal{S}_\mathbf{D}$) by

$$P^{i+1}(D) = P^1(P^i(D)).$$

Finally, let P^ω (called a datalog mapping) be the mapping from $\mathcal{S}_\mathbf{D}$ to $\mathcal{S}_\mathbf{D}$ defined (for each D in $\mathcal{S}_\mathbf{D}$) by

$$P^\omega(D) = \bigcup_{i=1}^{\infty} P^i(D).$$

The datalog mapping associated with a datalog program transforms each database to a least fixed point containing the database.

For simplicity, we do not distinguish between base predicates and derived predicates.

It is easily seen that P^i and P^ω are well defined, and $dom(P^i(D)) = dom(P^\omega(D)) = dom(D)$ for each i.

To illustrate the way a datalog mapping does its computation, consider the datalog program $P = \{R(x,y) \ \& \ R(y,z) \rightarrow R(x,z),$

$R(x,y) \rightarrow R(y,x)\}$ and the database $D = \{R(1,2), R(2,3), R(3,4)\}$. Then $P^1(D) = D \cup \{R(1,3), R(2,4), R(2,1), R(3,2), R(4,3)\}$, $P^2(D) = \{R(i,j) \mid 1 \leq i,j \leq 4 \text{ and } (i,j) \neq (4,1)\}$, and $P^\omega(D) = P^3(D) = \{R(i,j) \mid 1 \leq i,j \leq 4\}$.

Throughout the rest of the paper, we shall not mention the database scheme under consideration. Rather, it can be understood from the context. Hence, by the phrase "all databases" we mean all databases over some underlying database scheme.

For later usage, we now present the notion of logical implication and equivalence.

Definition. Let P and Q be datalog programs. Then P is said to logically imply Q, written $P \vdash Q$, if $D = P^\omega(D)$ implies $D = Q^\omega(D)$ for all D. If $P \vdash Q$ and $Q \vdash P$, then P and Q are said to be logically equivalent (denoted $P \equiv Q$).

The condition of $D = P^\omega(D)$ is sometimes referred to as "D satisfies P" or "D is a model of P" [Sagi].

3 Decomposition of databases

The central idea of the paper is to decompose databases so that the computation of a program is achieved by unioning its independent evaluations on the subdatabases. In this section we discuss what a decomposition is and what properties it should have.

We shall describe a decomposition using a function f from databases to finite sets of databases. For each database D, the elements in $f(D)$ are the subdatabases in the decomposition. The desired equation to ensure distributed processibility of a program P is

$$P^\omega(D) = \bigcup_{D' \text{ in } f(D)} P^\omega(D').$$

A number of parameters can be used in the decomposition of databases. One parameter is the occurrence of domain values or constants. Of course, there can be more than one way to use this parameter. We examine one way in Section 4. Another parameter is the size of subdatabases in the decomposition. Again, there can be numerous directions to investigate decomposition using this parameter. For example, the subdatabases can be bounded by a function of the size of the original database. We study one way by considering a particular function in Section 5. A third parameter is the relation names that occurs in facts. Clearly, all these parameters can be used separately or in combination.

Not all decompositions are desirable for the purpose of efficient distributed processing, e.g., a decomposition which always returns a finite set of databases including the input. Thus, restrictions on decompositions should be imposed to ensure a more efficient evaluation than in the single processor case. For example, there should be an D_0 such that $P^\omega(D') \neq P^\omega(D_0)$ for each D' in $f(D_0)$. In this paper, we shall propose two kinds of decompositions satisfying this requirement.

We shall not restrict ourselves to decompositions where each new fact is derived from exactly one subdatabase. Indeed, the decomposition considered in Section 4 satisfies this condition, whereas the one studied in Section 5 does not.

We conclude this section by noting a method for improving decompositions. Suppose f is a decomposition. Let f_{max} be the decomposition defined (for each database D) by

$$f_{max}(D) = \{D' \text{ in } f(D) \mid \text{no proper subset of } D' \text{ is in } f(D)\}.$$

Note that [Dong1] all datalog mappings P^ω are increasing, i.e., $P^\omega(D_1) \subseteq P^\omega(D_2)$ for all databases D_1 and D_2 such that $D_1 \subseteq D_2$. It is then clearly seen that f_{max} is a more efficient decomposition than f.

4 Disjoint distributivity

In this section we introduce and examine our first class of distributed-processible programs, namely the "disjoint-distributive." The major result characterizes such programs by a easily decidable syntactic condition called "connectivity."

Definition. Databases D_1 and D_2 are disjoint if no constant occurs in both D_1 and D_2, i.e., $dom(D_1) \cap dom(D_2) = \emptyset$.

Definition. A program P is disjoint distributive if $P^\omega(D_1 \cup D_2) = P^\omega(D_1) \cup P^\omega(D_2)$ for all disjoint databases D_1 and D_2.

We shall give examples of disjoint-distributive and nondisjoint-distributive programs later.

Clearly, the decomposition for disjoint-distributive programs returns all the disjoint subdatabases. Thus, suppose P is a disjoint-distributive program and D is a database. Then the computation of $P^{\omega}(D)$ can be carried out in parallel when D is partitioned into disjoint subdatabases. Furthermore, finding this partition of a database (which corresponds to the computation of the connected components in graph theory) is very efficient, namely, linear [AHU]. Hence the speed-up by using parallelism will not be wiped out due to the overhead.

In order to present our major result of this section, we need the following notion of "connectivity" as it pertains to variables, rules, and programs.

Definition. For each rule r and variables x and y occurring in $B(r)$, x and y are connected in r if (a) either x and y both occur in a common formula in $B(r)$ or (b) there is a variable z such that x and z are connected in r and y and z are connected in r. A rule r is connected if each pair of variables occurring in $B(r)$ are connected in r. A program P is connected if each rule in P is connected.

For instance, the program in Example 1 is connected, but the program $\{President(x) \& Boss(y,z) \rightarrow Boss(x,z)\}$ is not. (Actually, each program having only elementary chain rules [UlVG] is connected. And the program in Example 1 is an elementary chain rule program.)

The property of disjoint distributivity is algebraic in nature, and appears difficult to verify. On the other hand, the attribute of connectedness is syntactic, and thus easy to detect. Our next result shows these two conditions are essentially equivalent. Before the result can be stated, the concept of "nonredundancy" is needed.

Definition. A program P is nonredundant if (a) no rule can be removed, i.e., $P \not\equiv (P - \{r\})$ for each r in P, and (b) no formula can be removed, i.e., for each $k \geq 2$, $r = A_1 \& \cdots \& A_k \rightarrow H$ in P and i $(1 \leq i \leq k)$, $P \not\equiv (P \cup \{r_0\} - \{r\})$, where $r_0 = A_1 \& \cdots \& A_{i-1} \& A_{i+1} \& \cdots \& A_k \rightarrow H$. P is redundant if it is not nonredundant.

Theorem 1 (Characterization by Connectivity) *A nonredundant program P is disjoint distributive iff it is connected.*

Proof. Consider the "if." Suppose P is connected. Let D_1 and D_2 be disjoint databases. It suffices to show that $P^1(D_1 \cup D_2) = P^1(D_1) \cup P^1(D_2)$. [Indeed, it was shown [Dong1] that there exists an integer n such that $P^{\omega}(D) = P^n(D)$ for each D in $\{D_1, D_2, D_1 \cup D_2\}$. For each i $(1 \leq i \leq n)$, $P^i(D_1)$ and $P^i(D_2)$ are disjoint since rules do not introduce new constants. By repeated applications of $P^1(D_1 \cup D_2) = P^1(D_1) \cup P^1(D_2)$, it follows that $P^n(D_1 \cup D_2) = P^n(D_1) \cup P^n(D_2)$. Therefore, $P^{\omega}(D_1 \cup D_2) = P^n(D_1 \cup D_2) = P^n(D_1) \cup P^n(D_2) = P^{\omega}(D_1) \cup P^{\omega}(D_2)$. Hence P^{ω} is disjoint distributive.] Clearly, $P^1(D_1) \cup P^1(D_2) \subseteq P^1(D_1 \cup D_2)$. To establish the reverse containment, let $R(a_1, \cdots, a_m)$ be in $P^1(D_1 \cup D_2)$. It suffices to assume that $R(a_1, \cdots, a_m)$ is not in $D_1 \cup D_2$. Then there exist a rule $r = R_1(x_{11}, \cdots, x_{1n_1}) \& \cdots \& R_k(x_{k1}, \cdots, x_{kn_k}) \rightarrow R(y_1, \cdots, y_m)$ in P and an assignment τ satisfying the following two conditions: (α) $\tau(y_i) = a_i$ for each i $(1 \leq i \leq m)$; and (β) $\tau(R(x_{i1}, \cdots, x_{in_i}))$ is in $D_1 \cup D_2$ for each i $(1 \leq i \leq k)$. Without loss of generality, we may assume that $\tau(x_{11})$ occurs in D_1. Using (β) and the fact that r is connected, it is easily seen that $\tau(x_{ij})$ occurs in D_1 for each i and j $(1 \leq i \leq k$ and $1 \leq j \leq n_i)$. By (β) and the fact that D_1 and D_2 are disjoint, it follows that $\tau(R(x_{i1}, \cdots, x_{in_i}))$ is in D_1 for each i $(1 \leq i \leq k)$. Hence $R(a_1, \cdots, a_m) = \tau(R(y_1, \cdots, y_m))$ is in $P^1(D_1) \subseteq P^1(D_1) \cup P^1(D_2)$. Since $R(a_1, \cdots, a_m)$ is arbitrary, $P^1(D_1 \cup D_2) \subseteq P^1(D_1) \cup P^1(D_2)$.

To verify the "only if," suppose P is not connected. Then there exists a disconnected rule r in P. Let τ be a one-to-one assignment and $D = \{\tau(A) \mid A \text{ in } B(r)\}$. Since r is not connected, there exist two disjoint nonempty subsets B_1 and B_2 of $B(r)$ such that $B_1 \cup B_2 = B(r)$ and variables occurring in B_1 are not connected with variables occurring in B_2. Let $D_i = \{\tau(A) \mid A \text{ in } B_i\}$ for $i = 1, 2$. Clearly, D_1 and D_2 are disjoint and $D = D_1 \cup D_2$. Since P is nonredundant, $\tau(H(r))$ is not in $P^{\omega}(D_1) \cup P^{\omega}(D_2)$ [Sagi]. However, $\tau(H(r))$ is in $P^{\omega}(D)$. Thus, $P^{\omega}(D_1) \cup P^{\omega}(D_2) \neq P^{\omega}(D)$, and P is not disjoint distributive. □

Using Theorem 1, we now demonstrate examples of disjoint-distributive and nondisjoint-distributive programs. Since the program in Example 1 is connected, it is disjoint distributive by Theorem 1. (Using the comment given earlier, all elementary chain rule programs are disjoint distributive.) On the other hand, the program $\{President(x) \& Boss(y,z) \rightarrow Boss(x,z)\}$ is disconnected and nonredundant, and is thus not disjoint distributive by Theorem 1.

We now consider the nonredundancy condition in Theorem 1. For the "if" part, this condition can be removed by the proof of the theorem. For the "only if" part, however, this condition cannot be removed. Indeed, let $P = \{R(x,y)\&S(y,z)\&R(u,v) \rightarrow T(x,z)\}$. Then P is not connected. Let $Q = \{R(x,y)\&S(y,z) \rightarrow T(x,z)\}$. Then Q is connected and nonredundant, and thus disjoint distributive by Theorem 1. It can be verified that $P \equiv Q$, thus P is disjoint distributive.

Clearly, a rule is connected iff the corresponding (hyper)graph for its body is a connected graph. (For each rule body, the corresponding graph has a node for each variable, and has an edge $\{u_1, \cdots, u_m\}$ for each formula $R(u_1, \cdots, u_m)$.) Thus, it is easily decidable whether a program is connected [AHU]. By results in [Sagi], there is an algorithm which transforms a program into a logically equivalent nonredundant program. By Theorem 1, we get:

Proposition 1 (Decidability) *It is decidable whether a program is disjoint distributive.*

We conclude this section with an application of Theorem 1. Suppose P is a connected program. By the argument in the previous paragraph, P is disjoint distributive. Let Q be a nonredundant program which is logically equivalent to P. Then $Q^\omega = P^\omega$, and thus Q is disjoint distributive. By Theorem 1, Q is connected. Therefore we get the following invariance property of connectivity:

Corollary 1 *All nonredundant programs logically equivalent to a connected program are connected. Consequently, for all nonredundant logically equivalent programs P_1 and P_2, either both P_1 and P_2 are connected, or both are disconnected.*

It is open whether the number of connected components in the rule bodies are the same among logically equivalent nonredundant programs.

5 Bounded distributivity

In this section we introduce and scrutinize another class of distributed-processible programs, i.e., the "bounded-distributive." Intuitively, a program is in this class if its effects on a database is the union of its effects on all subdatabases with a fixed size. The major results characterize a program in this class by bounded recursiveness, by equivalence to a 1-bounded-recursive program without scratch paper, and by constant parallel complexity respectively.

We start this section with the notion of bounded distributivity.

Definition. A program P is ℓ-bounded distributive (ℓ a positive integer) if, for each database D with at least ℓ facts, $P^\omega(D) = \cup P^\omega(D_\ell)$, where the union is over all subdatabases with exactly ℓ facts. A program is bounded distributive if it is ℓ-bounded distributive for some ℓ.

Thus, the decomposition for bounded distributive programs returns all subdatabases with a fixed size.

We shall show that the program in Example 2 is bounded distributive, whereas the transitive closure program is not.

A program having only rules with one formula in its body is called *single-body* [Dong1]. Clearly, all single-body programs are 1-bounded distributive.

Similar to bounded distributivity, we call a program P *ℓ-pseudo-bounded distributive* if, for each database D with at least ℓ facts, $P^\omega(D) = \cup P^\omega(D_\ell)$, where the union is over all subdatabases with *at most* ℓ facts. Since datalog mappings are increasing, it is easily seen that pseudo-bounded distributivity is equivalent to bounded distributivity.

To present our first characterization theorem, the notion of bounded recursion is needed.

Definition. A program P is ℓ-bounded recursive (ℓ a positive integer) if $P^\omega = P^\ell$; and P is bounded recursive if it is ℓ-bounded recursive for some ℓ.

Another commonly used definition of bounded recursiveness is in terms of the equivalence to a nonrecursive program using "scratch paper" relations [NaSa]. In [Ioan] bounded recursiveness was called "uniform boundedness" and a graph characterization was given. Several other authors [GMSV, NaSa, Vard] studied the class of bounded-recursive programs with respect to decision problems. In general, such programs are preferred for their efficient evaluations.

Our first characterization theorem of bounded distributivity encompasses two equivalences, the first bounded recursion, and the second equivalence to a 1-bounded-recursive program (without scratch papers).

Theorem 2 (Characterization by Bounded Recursion) *For each program P, the following conditions are all equivalent:*

(a) P is bounded distributive;

(b) P is bounded recursive; and

(c) There exists a 1-bounded-recursive program Q using only relation names occurring in P such that $P^\omega = Q^1$.

Proof. Consider (a) implies (b). Suppose P is bounded distributive. Then there exists an integer ℓ such that for each database D with at least ℓ facts, $P^\omega(D) = \cup P^\omega(D_\ell)$, where the union is over all subdatabases with ℓ facts. Let k be the number of relation names occurring in P, j the maximum of the arities of the relation names occurring in P, and $n = k(j\ell)^j$. Let D_ℓ be a database with ℓ facts. Then there are at most $j\ell$ constants occurring in D_ℓ, and at most n facts in $P^\omega(D_\ell)$. Therefore, $P^\omega(D_\ell) = P^n(D_\ell)$. Let D be a database with at least ℓ facts. Then

$$\begin{aligned} P^\omega(D) &= \cup P^\omega(D_\ell) \\ &= \cup P^n(D_\ell), \text{ since } P^\omega(D_\ell) = P^n(D_\ell) \\ &\qquad\qquad \text{for each } D_\ell \\ &\subseteq \cup P^n(D) \\ &= P^n(D) \\ &\subseteq P^\omega(D). \end{aligned}$$

Thus, $P^\omega = P^n$, i.e., P is bounded recursive, and (b) is proven.

Now consider "(b) implies (c)." Suppose P is bounded recursive. Then there exists an integer n such that $P^\omega = P^n$. If $n = 1$, then there is nothing to prove. Suppose $n > 1$. Intuitively, we shall use all "resolution trees" of height n or less as the rules in the desired program. (This approach increases the program size considerably, and thus not practical.) Formally, let $m = max\{\#(B(r)) \mid r \text{ in } P\}$ and $\ell = \Sigma_{i=0}^n m^i$. We call two sets S_1 and S_2 of formulas *isomorphic* if there is a one-to-one function h from variables to variables such that $S_2 = \{h(F) \mid F \text{ in } S_1\}$. Clearly, among all nonempty finite sets of formulas using relation names occurring in P and having cardinality at most ℓ, there are only a finite number of nonisomorphic ones. Let $S_1, \cdots, S_k$ be all such sets. For each i $(1 \leq i \leq k)$, let B_i be the conjunction of all formulas in S_i. Let[1] $Q = \{B_i \rightarrow F \mid F \text{ in } P^\omega(S_i) - S_i \text{ and } 1 \leq i \leq k\}$. Then it can be shown that $Q^1 = P^\omega$. (Note that each "resolution tree" constructible using P is isomorphic to one rule in Q.) Thus (c) is verified.

Finally, consider "(c) implies (a)." Suppose there exists some 1-bounded-recursive program Q such that $P^\omega = Q^1$. Let $\ell = max\{\#(B(r)) \mid r \text{ in } Q\}$. Let D be an arbitrary database containing at least ℓ facts. Then $P^\omega(D) = Q^1(D) = \cup Q^1(D_\ell) = \cup P^\omega(D_\ell)$. That is, P is ℓ-bounded distributive. □

We note that if an integer ℓ such that P is ℓ-bounded distributive is effectively computable, then so is an integer n such that $P^\omega = P^n$; and conversely.

Using Theorem 2, we now demonstrate examples of bounded-distributive and nonbounded-distributive programs. The program in Example 2 is bounded recursive, and thus bounded distributive by Theorem 2. The transitive closure program is not bounded recursive, and thus not bounded distributive by the same theorem.

We now present the characterization of bounded distributivity by constant parallel complexity. For this result, we assume that a database is given as a sequence in shared memory, the access to the sequence is random, and the number of facts is provided as part of the input.

Theorem 3 (Characterization by Constant Parallel Complexity) *A program P is bounded distributive iff it has constant parallel complexity with a polynomial number of processors.*

Proof For the "only if" direction, suppose P is bounded distributive. Then there exists an integer ℓ such that P is ℓ-bounded distributive. Clearly, the bottom-up evaluation algorithm for P computes $P^\omega(D_0)$ in constant time, say C, for each database D_0 containing ℓ or fewer facts. Suppose we have an infinite number of processors, numbered from 1 to infinity. (Only a polynomial number of them is needed for each database in terms of the database size.) For each processor m, there is an associated vector $(a_{m1}, \cdots, a_{mr})$ generated by the following algorithm:[2]

```
Input: An integer $m \geq 1$.
Output: A vector representing a $\ell$-element combination of integers.
1. Let $a_j = j$ for each $j$, $1 \leq j \leq \ell$; $MaxGenerated = 1$.
```

[1] Here we assume the mapping is defined over sets of formulas just like it is defined over sets of facts.

[2] The segment inside the else part in 3 is obtained by modifying an algorithm in [Even].

```
2. If $MaxGenerated = m$ then stop, and $(a_1, \cdots, a_\ell)$ is the output.
3. MaxGenerated = MaxGenerated + 1;
    If $a_1 = a_\ell - \ell + 1$, then let $a_\ell = a_\ell + 1$ and $a_j = j$ for each $j < \ell$
    else find the largest $j$ such that $a_{j+1} \neq a_j + 1$ and let $a_{j+k} = a_j + k + 1$ for $0 \leq k < \ell - j$.
4. Goto 2.
```

Vector Generating Algorithm

Clearly, this algorithm can be modified to generate all vectors for all integers $1 \leq m \leq n$ for each integer n. This algorithm is of polynomial complexity in terms of n. Thus, the "circuit" needed for the processors can be set up efficiently.

The parallel algorithm for processor m ($m \geq 1$) is as follows:

```
Input: A database $D$ in the form of a sequence and number $NoFacts$ of facts in $D$.
Output: All facts in $P^\omega(D)$.
If the first value in the $m^{th}$ vector is larger than $NoFacts$ then do nothing
    else get the facts available according to the vector, and process them using the bottom-up evaluation.
```

Parallel Algorithm for Processor m

Let D be an arbitrary database. If D contains less than ℓ facts, then the 1st processor computes $P^\omega(D)$. If D contains at least ℓ facts, then the m^{th} processor picks up the m^{th} combination of ℓ facts from D and processes it. The above algorithm is correct since $P^\omega(D) = \cup P^\omega(D_\ell)$ for each database D with ℓ or more facts, where the union ranges over subdatabases of D containing exactly ℓ facts. Clearly this algorithm executes in time $O(C)$, that is, constant time.

For the "if" direction, suppose P has a constant parallel complexity. Thus, there exists a parallel algorithm which when given a database as a sequence and the number of facts in D as input, will compute $P^\omega(D)$ in constant time, say ℓ. It is clear that each processor m cannot read more than ℓ facts, say D^m, from D. Let $Processor_m(D^m)$ denote the result of processor m operated on D^m. Clearly, $Processor_m(D^m) \subseteq P^\omega(D^m)$. Thus, $P^\omega(D) = \cup_m Processor_m(D^m) \subseteq \cup_m P^\omega(D^m) \subseteq \cup P^\omega(D_\ell)$. Hence $P^\omega(D) = \cup P^\omega(D_\ell)$. □

Note that we ignored the cost of union of the facts from the processors.

We now apply our characterization results to bounded recursiveness. Suppose we are given a bounded-recursive program P. By definition, the evaluation of P on each database D can be achieved by applying P^1 a uniformly bounded number of times. However, such measure of complexity is not satisfactory, since it does not take into account the number of rule firings and the number of fact searches. Considering such firings and searches, neither the serial complexity nor the parallel complexity is known. This lack of knowledge may be due to the complications from the interactions among the facts, or from the number of fruitless searches. Using our earlier theorems, we now present two useful results providing tight upper bound of such complexities. In particular, the first result characterizes bounded-distributive programs by constant parallel complexity. The second result says that the serial complexity (including search for facts) is bounded by a polynomial of the input size.

Corollary 2 (Characterization by Constant Parallel Complexity) *A program is bounded recursive iff it has constant parallel complexity (including search for facts).*

By Corollary 2 and the fact that there are only a polynomial number of subdatabases of fixed size, we get:

Corollary 3 (Polynomial Number of Fact Searches) *Each bounded-recursive program has polynomial serial complexity (including search for facts).*

We conclude this section by noting the following: Bounded distributivity and disjoint distributivity can be combined for more efficient distributed processing. Indeed, suppose P is both bounded distributive and disjoint distributive. Then we can decompose each database into disjoint subdatabases bounded by a fixed size. Clearly, in most cases, this decomposition has fewer subdatabases, which implies enhanced efficiency.

6 Conclusions

In this paper we considered the issue of distributed or parallel processing of datalog programs. We focused on decomposing databases into a number of subdatabases such that the computation of a program on a database can be achieved by *unioning* its *independent evaluations* on the subdatabases. Such decomposition translates readily to distributed processing without communication, and thus implies a significant speed-up of the evaluation process. We introduced and examined two kinds of distributed-processible programs according to the properties of database decomposition, namely the disjoint-distributive and the bounded-distributive. We characterized disjoint-distributive programs by an easily decidable syntactic condition of connectivity; and characterized bounded-distributive programs by bounded-recursion, 1-bounded recursion, and constant parallel complexity respectively. Our study has implications on bounded recursion as well.

A number of interesting questions are raised by this study. An important direction is to identify other classes of programs similar to the disjoint distributive and the bounded distributive but defined with different kind of decompositions. For example, one may want to consider decompositions which return all subdatabases bounded by a logarithmic of the input size. A second direction is to combine our approach with existing approaches like decomposing predicates [WoSi] and magic sets [BaRa]. Third, one may consider decomposition of databases which require some communication among processors. Fourth, our study also has applications to optimization of datalog program evaluation on a single processor. However, further investigation is needed to convert ideas on parallel evaluations into ideas on single processor evaluation.

Acknowledgement The author is grateful to S. Ginsburg, R. Hull, K. Kompella, and S. Kurtzman for helpful discussions. He is also thankful to the anonymous referees for suggestive comments which improved the presentation.

References

[AHU] Aho, A.V., Hopcroft, J.E., and Ullman, J.D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Publishing Company, 1974.

[BaRa] Bancilhon, F., and Ramakrishnan, R. An amateur's introduction to recursive query processing strategies. In *Proceedings of the ACM SIGMOD Conference*, 1986, 16-52.

[CoKa] Cosmadakis, S.S., and Kanellakis, P.C. Parallel evaluation of recursive rule queries. In *Proceedings of the ACM Symposium on Principles of Database Systems*, 1986, 280-293.

[CoWo] Cohen, S.R., and Wolfson, O. Why a single parallelization strategy is not enough in knowledge bases. In *Proceedings of PODS*, 1989.

[Dong] Dong, G. On the composition and decomposition of datalog program mappings. In *Proceedings of 2nd ICDT*, Belgium, 1988, Lecture Notes in Computer Science No. 326 (M. Gyssens, J. Paredaens, and D. Van Gucht eds). To appear in a special issue of *Theoretical Comp Sci.*

[Dong1] Dong, G. On the composition and decomposition of datalog program mappings. Ph.D. Thesis, USC, 1988.

[Even] Even, S. *Algorithmic Combinatorics.* The Macmillan Company, New York, 1973, pp. 32.

[GMSV] Gaifman, H., Marison, H., Sagiv, Y., Vardi, M.Y. Undecidable optimization problems for database logic programs. In *Proceedings of 2nd IEEE Symposium on Logic in Computer Science*, Ithica, 1987, 106-115.

[Ioan] Ioannidis, Y.E. A time bound on the materialization of some recursively defined views. In *Proceedings of International Conference on Very Large Data Bases*, 1985.

[Ll] Lloyd, J.W. *Foundations of Logic Programming.* Springer-Verlag, 1984.

[Naug] Naughton, J. Data independent recursion in deductive databases. In *Proceedings of the ACM Symposium on Principles of Database Systems*, 1986, 267-279.

[NaSa] Naughton, J., and Sagiv, Y. A decidable class of bounded recursion. In *Proceedings of the ACM Symposium on Principles of Database Systems*, 1987, 227-236.

[Sagi] Sagiv, Y. Optimizing datalog programs, In *Proceedings of the ACM Symposium* *on Principles of Database Systems*, 1987, 349.

[UlVG] Ullman, J.D. and Van Gelder, A. Parallel complexity of logical query programs. In *Algorithmica* (1988) 3. pp. 5-42.

[Vard] Vardi, M.Y. Decidability and undecidability results for boundedness of linear recursive programs. In *Proceedings of the ACM Symposium on Principles of Database Systems*, 1988, 341-351.

[WoSi] Wolfson, O. and Silberschatz, A. Distributed processing of logic programs, In *Proceedings of the ACM SIGMOD Conference*, 1988, pp. 329.

[Wolf] Wolfson, O. Sharing the load of logic-program evaluations. To appear in the *Proceedings of the International Symposium on Databases in Parallel and Distributed Systems.* December, 1988.