University of Edinburgh

Department of Computer Science

Parallel Processing of Linear Recursive Datalog Queries using Transitive Closure Algorithms

4th Year Project Report

Thomas Zurek

Supervisor:

Peter Thanisch

June 3, 1992

Abstract

Datalog is a subset of Prolog and is used as a query language for deductive databases. The advantage of Datalog over query languages based on relational algebra is that it allows recursion. Normally relational query languages like SQL lack such mechanisms. The disadvantage is that recursive Datalog queries are expensive to process by the usual methods like bottom-up or top-down. This report investigates two approaches to improving the performance of Datalog query processing: replacing linear recursion by using (parallel) transitive closure algorithms and using the data parallelism that is inherent in a large number of database applications. Finally the results obtained in the first chapters of this report were used as the basis of an implementation of the query processing algorithm of Jagadish *et al.* on a Connection Machine CM-200 and performance results of this implementation are given.

Contents

Chapter 1

Introduction

1.1 Overview

Datalog is a query language for deductive databases. It is a 'subset' of Prolog, lacking some features such as function symbols in argument terms. Nevertheless it is more powerful than usual relational query languages like SQL as it allows recursion[1]. There is a good deal of interest in recursive query languages for various reasons. We mention just two motivating applications here.

Logic rules are processed by inference mechanisms of some expert systems. The knowledge base of such systems may incorporate a traditional database to store the facts which are accessed and used by the inference processor. This is one reason to have effective processing techniques for queries based on logical rules.

Many traditional database applications, such as posing queries about a part - subpart hierarchy, invoke viewing the data in tables as a directed graph and require a recursion mechanism. If the query language cannot express recursive queries then the recursion mechanism has to be in a host language that embeds queries. This can be inefficient as it effectively makes the optimization of query processing the responsibility of the applications programmer. Thus there is a motivation to developing a query language that can capture a broader class of queries.

To improve the performance of query processing relational database systems incorporate a query optimizer. Even for the class of nonrecursive queries, query optimization is an NP-hard problem [Ull 89]. Thus positive complexity results for first-order queries do not translate into efficient processing techniques. At the moment there is nothing like a query optimizer component for Datalog query processing although there are some approaches for optimizing. An overview is given in chapter 2 which also contains an introduction to Datalog and its relationship to relational algebra.

In this project we concentrate on a strict sub-class of recursive queries, namely the class

[1] We note that ad hoc recursion mechanisms have been incorporated in some implementations like Oracle's SQL*plus.

of *linear* recursive queries. Jagadish *et al.* have proposed a technique for processing this class of queries which involves replacing the linear recursion with some transitive closure and some relational algebra operations. The idea of this approach and the processing algorithm is presented in chapter 3. Jagadish *et al.* were not concerned with parallel processing. However, in the present project, we investigate a parallel implementation of this approach using the SIMD[2] Connection Machine.

Chapter 4 investigates some parallel transitive closure algorithms that are based on a boolean matrix representation of the graph corresponding to a database relation. The algorithms were implemented in C* on a Connection Machine CM-200 and the performance results are given. Some conclusion about their effectiveness and usefulness for different database relations can be drawn and were used in the implementation presented in chapter 5.

Apart from the implementation of the processing algorithm chapter 5 presents an evaluation of the performance results for different classes of queries. These results may be used in an optimizer for Datalog queries.

The next section will give a short introduction to the hardware architecture and the software environment to make the reader familiar with possibilities of the CM and the kind of parallelism that can be exploited.

1.2 Hardware Architecture and Software Environment

1.2.1 Hardware Architecture

The CM is a SIMD computer with 16384 processors connected by a 10-dimensional hypercube network. Each processor has a local memory of 256 kBits so the CM has 512 MBytes total memory. In one step each active processor executes the same instruction but on different data (i. e. SIMD) whereas inactive processors are idle. The successful use of the CM requires an algorithm that can exploit this data parallelism. The large number of processors, the underlying communication network and the parallel mass storage system (the 'Data Vault') make it suitable for processing large amounts of data.

The Data Vault has 10 GBytes of memory and a possible data transferring rate of 25 MBytes per second. It employs an array of 42 disk drives. Of these 42 drives, 32 hold data. Reading from and writing to these 32 drives means reading and writing on 32 drives in parallel. Having a fast secondary storage system is really important as the sizes of databases are growing with increasingly ambitious database modelling.

Data parallel computing associates one processor with each data element. Therefore parallel data structures are spread across the processors, with a single element stored in each processor's memory. When parallel data structures contain more elements than the system has processors,

[2] Single Instruction Multiple Data stream

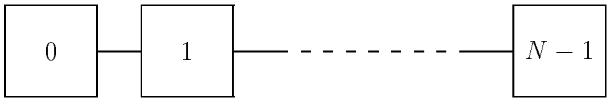


Figure 1.1: A linear array connection of N processors

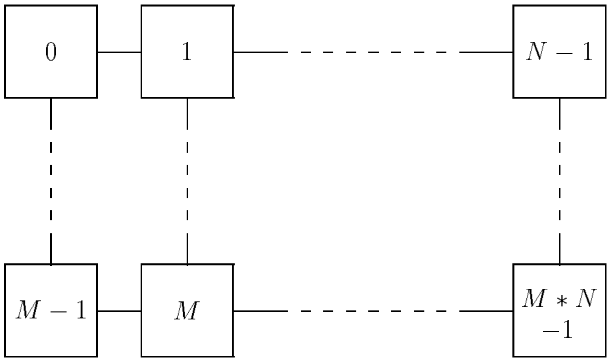


Figure 1.2: A matrix connection of $M \times N$ processors

the system operates in virtual processor mode, presenting the user with a large number of processors with a correspondingly smaller memory and performance. Each physical processor is made to simulate the appropriate number of virtual processors [CM 91].

1.2.2 C*

The C* language is an extension of C for the use with an SIMD machine. C* programs have normal sequential control flow so there are no new synchronization structures needed. The language construct that involves data parallelism is the notion of *parallel variables.* If x is a parallel variable then the components of x are allocated in each (virtual) processor, which implies that there is a set of virtual processors attached to a parallel variable. This set is called a *shape*, which not only defines the number of virtual processors but also the way in which they are organized. The simplest example is a one-dimensional shape A:

shape $[N]$ A;

In this case there are N virtual processors in a linear array connection (see figure 1.1). Another example is a two-dimensional shape B where the virtual processors are organized in an $M \times N$ matrix (see figure 1.2):

shape $[M][N]$ B;

A parallel variable can be declared by giving its type *and* its shape. For example

shape [65536] A;
int:A x;

declares a variable x of type integer to be attached to each of the 65536 virtual processors which means that each of the 16384 physical processors simulates 65536:16384 = 4 virtual processors. The related shape has to be activated by a *with* statement before the parallel variable can be referenced. For example the expression

$$
\begin{array}{l}
\text{with } (A) \\
\quad x = 1;
\end{array}
$$

assigns 1 to x in every virtual processor. Note that this operation takes one step. If only certain processors are expected to get a value for x then these processors are set to 'active' whereas all other processors are 'inactive'. This can be done using a *where* statement:

$$
\begin{array}{l}
\text{with } (A) \\
\quad \text{where (pcoord(0)} < 100 \text{)} \\
\quad\quad x = 1;
\end{array}
$$

assigns 1 to x in the processors 0, 1, 2, ..., 99. For an example of C* code see figure 4.8. The notion of a parallel variable and its handling are the essential extensions for achieving data parallelism. They were presented to make the reader familiar with the possibilities of parallelism on a Connection Machine.

Chapter 2

Datalog

This chapter will give an introduction to Datalog as a database query language. Section 2.2 shows the relationship between Datalog and the large family of query languages that are based on relational algebra. This relationship is relevant for subsequent chapters as we will follow an optimization technique that processes a Datalog query by using operators of relational algebra supplemented by a transitive closure operator. Finally section 2.3 gives an overview of optimization techniques for Datalog queries.

2.1 Introduction

Datalog is a subset of Prolog which is suitable for database systems. It lacks some features of Prolog and can be considered as an experimental recursive query language. It has a set-oriented semantics, i. e. we are interested in the set of all solutions for a Datalog program (query) and not only in one. The reader is assumed to be familiar with Prolog so we will concentrate on the main issues and the differences between Datalog and Prolog. For a more detailed introduction to Datalog see chapter 3 of [Ull 88].

Rules

A Datalog query comprises a collection of logical rules. A *rule* can be thought of as an if ...then ... clause: e. g. the rule

$$p(X_1, \ldots, X_n)\ :-\ q(Y_1, \ldots, Y_m),\ r(Z_1, \ldots, Z_l).$$

says that $p(X_1, \ldots, X_n)$ holds if $q(Y_1, \ldots, Y_m)$ and $r(Z_1, \ldots, Z_l)$ hold. $p(X_1, \ldots, X_n)$ is a *literal* and is called the *head* of the rule whereas $q(Y_1, \ldots, Y_m)$ and $r(Z_1, \ldots, Z_l)$ are also literals and form the *body* of the rule. The order of the literals in the body is of no importance: the declarative semantics of Prolog is used for the interpretation of Datalog queries. Rules are *nonrecursive* if the predicate symbol of the head does not appear in the body; otherwise they are *recursive*. The body may also be empty, in which case the rule is called a *fact*. n, m and l are the *arities* of the predicate symbols p, q and r respectively.

Arguments

One major difference between Prolog and Datalog is that only variables (arguments starting with a capital letter) and constants (arguments starting with a small letter or a number) and such constructs as lists and terms using function symbols, e. g. $s(s(s(0)))$ are not allowed as arguments. Database theory normally assumes finite relations but if function symbols were allowed it would be possible to define infinite relations. They would cause severe problems in implementing query optimization and processing techniques.

Predicate Symbols and Relations

The correspondence of relations and Datalog rules is that the underlying model of data for Datalog is essentially that of the relational model: predicate symbols correspond to relations so, e. g., if $p(a, b, c)$ holds then the tuple (a, b, c) is in the relation P. Predicates will be referred to by small letters and the corresponding relations by capitals (or names starting with a capital letter): for example *Flows_into* is the relation related to the predicate *flows_into*. One has to be careful when using attribute names to refer to the columns of the relations. In Datalog a column is referred to by its position among the arguments of a given predicate symbol. For example in $p(X, Y, Z)$ the variable X denotes the first component of some tuple in the relation P which is referred to as as \$1 ('position 1') or $\$X$ ('position of X').

Extensional and Intensional Database Relations

There are two ways relations can be defined in Datalog: firstly relations can be stored in a relational database in which case they are are called *extensional* database (EDB) relations. The other way is to define relations by logical rules; they are called *intensional* database (IDB) relations. It is assumed that each predicate symbol either refers to an EDB or an IDB relation, but not both. When defined non-recursively IDB relations correspond to *views* in the relational model but they are much more powerful especially in the case that an IDB relation is defined by a recursive rule.

Example 1:

$$tributary(A, B) \quad :- \quad flows\_into(A, B). \tag{2.1}$$

$$tributary(A, B) \quad :- \quad flows\_into(A, C), tributary(C, B). \tag{2.2}$$

(2.1) is a nonrecursive rule that says that a river A that flows into river B is a tributary of B. More formally it defines an IDB relation *Tributary* to hold all tuples of the EDB relation[1] *Flows_into*. (2.2) is a recursive rule because the predicate symbol *tributary* of the head reappears in the body of the rule. It says that a tuple (A, B) is in *Tributary* if there is a tuple (A, C) in *Flows_into* and if there is already a tuple (C, B) in *Tributary*. This follows the idea that each immediate tributary A of C, which is itself a tributary of B, is also a tributary of B.

The predicate symbol *tributary* appears only once in the body of (2.2). Therefore the rule is said to be *linear recursive*. Imagine the rule

$$tributary(A, B) \quad :- \quad tributary(A, C), tributary(C, B). \tag{2.3}$$

which could be used instead of (2.2). (2.3) is not linear recursive because its body contains the head predicate symbol twice.

Built-in Predicates

Apart from ordinary predicates (like *flows_into* and *tributary* in the last example) there are several built-in predicates, essentially the arithmetic comparison predicates $=$, $\leq$, $>$ etc. The built-in predicates represent relations that are not necessarily finite, e. g. it is impossible to represent '$<$' as a finite database relation. Therefore it is assumed that whenever a term like $X < Y$ appears in the body of a rule the variables are limited in their range by some ordinary predicate. This restriction makes sense because database operations on finite relations result in finite relations.

In general this constraint is satisfied by restricting Datalog rules to be 'safe'. A rule is *safe* if all variables appearing in the rule are 'limited'. A variable X is *limited* if

1. X appears in some ordinary predicate of the body,
2. $X = a$ or $a = X$ is a subgoal or
3. $X = Y$ or $Y = X$ is a subgoal and Y is limited

Example 2:

$$bigger(X, Y) \quad :- \quad X > Y. \tag{2.4}$$

$$bigger(X, Y) \quad :- \quad number(X),\ number(Y),\ X > Y. \tag{2.5}$$

[1] The Datalog query of the example does not necessarily imply that *Flows_into* is an EDB relation because it may be defined as an IDB relation by some other rules. The convention throughout this paper is that a relation is to be assumed an EDB relation if no rules are given to define it as an IDB relation.

$$can\_use(S,C) \;:-\; computer(C). \tag{2.6}$$

$$can\_use(S,C) \;:-\; student(S),\ computer(C). \tag{2.7}$$

- (2.4) is not safe because X and Y are not limited;
- (2.5) is safe;
- (2.6) is not safe because S is not limited;
- (2.7) is safe.

2.2 The Relationship between Datalog and Relational Algebra

There is a strong relationship between Datalog rules and expressions of relational algebra called *Datalog equations* [Ull 88]. A Datalog equation of a nonrecursive rule can be simply evaluated by the normal query processing techniques, whereas computing the meaning of a Datalog equation originating in a recursive rule (i. e. a recursive Datalog equation) is more difficult. We will show how Datalog equations can be derived from an arbitrary Datalog rule. See [Ull 88] for a more detailed discussion of evaluating Datalog equations.
Generally a Datalog rule r looks like this:

$$\begin{aligned} p(X_1^0,\ldots,X_{n_0}^0) \;:-\; & q_1(X_1^1,\ldots,X_{n_1}^1),\ \ldots,\ q_k(X_1^k,\ldots,X_{n_k}^k), \\ & built\text{-}in_1(V_1^1,\ldots,V_{m_1}^1),\ \ldots,\ built\text{-}in_h(V_1^h,\ldots,V_{m_h}^h). \end{aligned}$$

with

- $k \geq 0$,
- $h \geq 0$,
- for all $1 \leq i \leq k$: $n_i \geq 1$ and q_i is an ordinary predicate,
- for all $1 \leq i \leq h$: $m_i \geq 1$ and $built\text{-}in_i$ is a built-in predicate
- the arguments X_j^i and V_j^i are not necessarily distinct.

The Datalog equation E for r can be derived in the following way:

1. Derive the expression E_i for each subgoal $q_i(\ldots)$ according to the following instructions. They may be applied several times to a subgoal.

(a) For subgoals

$$q(X_1, \ldots, X_n)$$

and $X_1, \ldots, X_n$ are distinct variables[2]

$$E_i = Q(X_1, \ldots, X_n)$$

(b) For subgoals

$$q(X_1, \ldots, X_{s-1}, Y, X_{s+1}, \ldots, X_{t-1}, Y, X_{t+1}, \ldots, X_n)$$

and $X_1, \ldots, X_n, Y$ are distinct variables

$$E_i = \sigma_{\$s=\$t}(Q(X_1, \ldots, X_{s-1}, Y, X_{s+1}, \ldots, X_{t-1}, X_t, X_{t+1}, \ldots, X_n))$$

where $\$s$ and $\$t$ refer to the s-th and the t-th column of Q respectively.

(c) For subgoals

$$q(X_1, \ldots, X_{s-1}, a, X_{s+1}, \ldots, X_n)$$

and $X_1, \ldots, X_{s-1}, X_s, X_{s+1}, \ldots, X_n$ are distinct variables

$$E_i = \sigma_{\$s=a}(Q(X_1, \ldots, X_{s-1}, X_s, X_{s+1}, \ldots, X_n))$$

where $\$s$ refers to the s-th column of Q and a is a constant

and compute

$$E' = E_1 \bowtie \ldots \bowtie E_k$$

where $\bowtie$ is the natural join operator.

2. For the subgoals involving built-in predicates like $=$, $<$, $>$ etc. add a selection operation (one per built-in predicate), e. g. for the subgoal $X = Y$

$$E'' = \sigma_{\$X=\$Y}(E')$$

where $\$X$ and $\$Y$ are the positions (in E') of the columns denoted by X and Y. If there are no subgoals involving built-in predicates then

$$E'' = E'$$

[2] It is important to note that the Prolog style variables X_i are transfered to column names in relations. One has to keep track of these names because the relation may be involved for example in a join operation. In this case it is necessary to identify the join attributes / columns.

3. E'' may have more columns than the head-literal of r has arguments. The columns corresponding to variables that are not among the arguments of the head literal must be omitted. For example if only columns 1, 2 and 4 of E'' are needed then do a projection operation to get

$$E''' = \pi_{\$1,\$2,\$4}(E'')$$

otherwise

$$E''' = E''$$

4. If there are repeated variables among the arguments of the head-literal it can be necessary to duplicate columns of the resulting E''' of 3. by using projection, selection and cartesian product operations. For example if $p(X, X, Y)$ is the head-literal and 3. results in $E'''(X, Y)$ then

$$E = \sigma_{\$1=\$2}(\pi_{\$1}(E'''(X,Y)) \times E'''(X,Y))$$

otherwise

$$E = E'''$$

Example 3: a. $grandparent(X, Y)\ :-\ parent(X, U),\ parent(U, Y).$

$$\begin{aligned}
E_1(X,U) &= Parent(X,U)\\
E_2(U,Y) &= Parent(U,Y)\\
E'(X,U,Y) &= Parent(X,U) \bowtie Parent(U,Y)\\
E''(X,U,Y) &= E'(X,U,Y)\\
&= Parent(X,U) \bowtie Parent(U,Y)\\
E'''(X,Y) &= \pi_{\$1,\$3}\,(E''(X,U,Y))\\
&= \pi_{\$1,\$3}\,(Parent(X,U) \bowtie Parent(U,Y))\\
&= E(X,Y)\\
Grandparent(X,Y) &= \pi_{\$1,\$3}\,(Parent(X,U) \bowtie Parent(U,Y))
\end{aligned}$$

b. $tributary(A, B)\ :-\ tributary(A, C),\ flows\_into(C, B).$

$$\begin{aligned}
E_1(A,C) &= Tributary(A,C)\\
E_2(C,B) &= Flows_into(C,B)\\
E'(A,C,B) &= E_1 \bowtie E_2\\
&= Tributary(A,C) \bowtie Flows_into(C,B)\\
E''(A,C,B) &= E'(A,C,B)
\end{aligned}$$

$$
\begin{aligned}
E'''(A,B) &= \pi_{\$1,\$3}\,(E''(A,C,B)) \\
&= \pi_{\$1,\$3}\,(\mathit{Tributary}(A,C) \bowtie \mathit{Flows\_into}(C,B)) \\
&= E(A,B) \\
\mathit{Tributary}(A,B) &= \pi_{\$1,\$3}\,(\mathit{Tributary}(A,C) \bowtie \mathit{Flows\_into}(C,B))
\end{aligned}
$$

The resulting Datalog equation of example 3 a shows that evaluating a nonrecursive rule by relational operators is fairly easy and can be done by the usual query processing techniques. One has only to pay attention to the order in which two or more nonrecursive rules are processed: an IDB relation has to be computed before it is involved in the processing of another IDB relation. For example in

$$q(X) \quad :- \quad r_1(X,Y),\ r_2(X,Z). \tag{2.8}$$

$$p(U,V) \quad :- \quad r_3(U,V,X),\ q(X). \tag{2.9}$$

(2.8) must be evaluated before (2.9) because Q is an IDB relation that defines P. The order in which the rules have to be processed is determined by the *dependency graph* of the rules. See [Ull 88] for details.

Example 3 b results in a recursive Datalog equation. In general recursive equations can be much more difficult to process. A naive technique is to start evaluation by using an initial value for the recursive relation[3] and to evaluate the relational expression (given by the Datalog equation). The result will be used for another evaluation of the expression and so on. Computation stops when no more tuples are added. At this point the relation *Tributary* is a *fixed point* of the Datalog equation

$$\mathit{Tributary(A,B)} = \mathit{Tributary(A,C)} \bowtie \mathit{Flows\_into(C,B)}.$$

(see example 3 b. This naive technique is quite similar to a bottom-up evaluation of the corresponding recursive rule involving the exit rule as an initial value. Normally database relations hold a large number of tuples so this very expensive technique is of no practical use. Therefore a lot of research has been done on techniques to optimize recursive Datalog queries.

[3] This initial value is usually given by one or more (nonrecursive) exit rules for the recursive rule. E. g. an exit rule for the recursive rule of example 3 b is $\mathit{tributary}(A,B)\ :-\ \mathit{flows\_into}(A,B)$ so the relation *Flows_into(A,B)* is the initial value in this case.

2.3 Approaches to the Optimization of Datalog Queries

2.3.1 Overview

There are two major approaches to optimizing Datalog queries:

Query Transformation Techniques: These techniques are quite similar to optimization techniques for queries expressed in relational algebra which replace the original expression with a new relational algebra expression that can be evaluated more cheaply. The goal is to rewrite Datalog rules to get an equivalent query which can be processed at a lower 'cost', in terms of CPU time or number of disk I/O operations.

Transitive Closure Techniques: These techniques focus on linear recursive Datalog queries. The goal is to transform a recursive Datalog rule so that it can be processed by using a transitive closure operation. This approach can exploit special algorithms for computing transitive closures efficiently.

2.3.2 Query Transformation Techniques

The difficulty of these techniques is that the *equivalence problem* for Datalog rules is undecidable, i. e. there can be no algorithms that can determine whether an arbitrary pair of queries produces the same result [Shmueli]. Sagiv investigated properties of a stricter form of equivalence namely *uniform equivalence*: two Datalog queries are uniformly equivalent if they produce the same result for all possible assignments of initial relations to all predicates. Uniform equivalence is decidable and it implies equivalence [Sag 88].

The cost of evaluating a Datalog query is related not only to the depth of the recursion but also to the structure of each rule: the number of (computationally) expensive joins is related to the number of literals in the body of a rule; so one goal to optimize a query is to detect redundant literals in the body of a rule. This technique uses the notion of uniform equivalence.

Another computational advantage can be achieved by replacing nonlinear with linear recursion (see [Sar 89], [Zhang]). Linear recursive queries can be evaluated by using transitive closure techniques (see next section).

2.3.3 Transitive Closure Techniques

The idea of these approaches is to translate a recursive Datalog query into an expression of relational algebra supplemented by a transitive closure operator. The notion is to replace recursion by a transitive closure.

Example 4: The linear recursive query

$p(X,Z)$	$:-$	$q(X,Z).$	Exit Rule
$p(X,Z)$	$:-$	$p(X,Y),\ q(Y,Z).$	Linear Recursive Rule

computes in fact the transitive closure of the relation Q. This becomes more obvious by looking at a bottom-up evaluation of the query – step i shows which tuples qualify for the IDB relation P if the linear recursive rule is applied i times:

Step 0:	$p(X,Z)$	$:-$	$q(X,Z)$.
Step 1:	$p(X,Z)$	$:-$	$q(X,Y_1),\ q(Y_1,Z)$.
Step 2:	$p(X,Z)$	$:-$	$q(X,Y_2),\ q(Y_2,Y_1),\ q(Y_1,Z)$.
Step 3:	$p(X,Z)$	$:-$	$q(X,Y_3),\ q(Y_3,Y_2),\ q(Y_2,Y_1),\ q(Y_1,Z)$.

...

The union of the sequences of q-literals is the transitive closure Q^* of Q so

$$P(X,Z) = Q^*(X,Z)$$

Jagadish *et al.* show that every linear recursive Datalog query can be evaluated by using a suitable transitive closure algorithm to eliminate recursion (see [Jag 87]). This can be done by analysing the *substitution graph* of the linear recursive rule. This technique forms the basis of the implementation described in subsequent chapters of this report.

An important point regarding the approach of [Jag 87] is that linear recursion cannot be effectively replaced by a data-independent number (i. e. a number that can be derived from the rules) of transitive closure operations. This can be seen by the following example.

Example 5: The linear recursive query

$p(X,Z)$	$:-$	$e(X,Z)$.	Exit Rule
$p(X,Z)$	$:-$	$p(X,Y),\ q(X,Y,Z)$.	Linear Recursive Rule

is not as easy to process as the example above:

Step 0:	$p(X,Z)$	$:-$	$e(X,Z)$.
Step 1:	$p(X,Z)$	$:-$	$e(X,Y_1),\ q(X,Y_1,Z)$.
Step 2:	$p(X,Z)$	$:-$	$e(X,Y_2),\ q(X,Y_2,Y_1),\ q(X,Y_1,Z)$.
Step 3:	$p(X,Z)$	$:-$	$e(X,Y_3),\ q(X,Y_3,Y_2),\ q(X,Y_2,Y_1),\ q(X,Y_1,Z)$.

...

The sequences of q-literals in steps 1, 2, 3, ... represent no single transitive closure but a union of several transitive closures: one for each value of the X-attribute in the relation E. Therefore the number of values for X is determined by the relation E, so the number of transitive closures depends on the actual data and is therefore data-dependent. Let us assume that there are two values $\{a,b\}$ for X in E. Then the resulting relation P can be computed by

$$P(X,Z) \quad = \quad E(X,Z) \bowtie$$

$$
\begin{aligned}
&(\{a\} \times [\pi_{\$2,\$3}(\sigma_{\$1=a}(Q(X,Y,Z)))]^* \\
&\cup \{b\} \times [\pi_{\$2,\$3}(\sigma_{\$1=b}(Q(X,Y,Z)))]^*)
\end{aligned}
$$

where $*$ denotes the transitive closure operator.

Youn *et al.* use so called *I-graphs*[4] to design an evaluation plan for the query [You 92]. I-graphs are very similar to substitution graphs and keep track not only of the behaviour of the recursive literals of the rule but also of that of the nonrecursive literals in the body.

Another approach is used by Grahne *et al.* which derives regular expressions from the Datalog query and uses finite automata techniques to get evaluation plans for this query [Gra 87].

As mentioned in the previous section transitive closure techniques can be easily combined with some query transformation techniques: the method of [Jag 87] is applicape to *every* linear recursive query so it is quite useful to involve approaches to rewrite multiple recursive to equivalent linear recursive queries.

Ullman suggests another, rather different approach in [Ull 89] using a graph representation of the underlying Datalog program. The answer for a linear recursive query can be given by computing one single transitive closure of this graph. In contrast to all the other techniques mentioned so far no operations of relational algebra are used and only one transitive closure has to be computed independent of the data. Ullman's approach is not of practical use because it generates a graph with $O(n^k)$ nodes where n is the number of all possible values of the database that can appear as arguments of IDB predicates and k is the largest arity of an IDB predicate. In practice n may be very large and the graph might need much more memory than the actual database. This can be seen by the following example.

Example 6: The query

$$
\begin{aligned}
p(X_1,\ldots,X_n,Z) \quad &:- \quad e(X_1,\ldots,X_n,Z). && (2.10)\\
p(X_1,\ldots,X_n,Z) \quad &:- \quad p(permutation(X_1,\ldots,X_n),Z),\ q(Y,Z). && (2.11)
\end{aligned}
$$

is processed by Ullman's algorithm in the following way: there is a domain D defined by

$$D = \{ \text{ all possible values of } X_1,\ \ldots,\ X_n,\ Y,\ Z \text{ in } (2.11) \ \}$$

The set V of vertices of the graph is given by

[4] These were introduced by Ioannidis in [Ioa 85].

$$V = \underbrace{D \times D \times \cdots \times D}_{n+1} = D^{n+1}$$

where $n+1$ is the arity of predicate p. The size $|V|$ of V gives the number of vertices in the graph:

$$|V| = |D|^{n+1}$$

Assume now

$$\begin{aligned}
n &= 4 \\
\text{number of values for each } X_i \text{ in } E &= 20 \\
\text{number of values for } Z \text{ in } E &= 100 \\
\text{number of values for each } Y, Z \text{ in } Q &= 100
\end{aligned}$$

and the values of Z in E and that of Y, Z in Q to be the same. Then

$$\begin{aligned}
|D| &= 20 + 20 + 20 + 20 + 100 &&= 180 \\
|V| &= |D|^{n+1} = |D|^5 = 180^5 &&\approx 189 \cdot 10^9
\end{aligned}$$

The number of edges is unpredictable and depends on the permutation of the X_i in the antecedent of (2.11). In the worst case all elements of V are involved so the resulting graph consists of approx. 189 billion vertices – invoked by possibly small relations E and Q. Undoubtely computing the transitive closure of this graph exceeds usual memory and CPU resources.

Chapter 3

Optimization of Linear Recursive Queries

This chapter will present the optimization technique that was used for the implementation discussed in chapter 5. The approached is based on the paper [Jag 87] of Jagadish *et al.* Section 3.1 defines the class of Datalog queries that will be focused on. Section 3.2 will show the notion of replacing linear recursion by one or more transitive closures. Section 3.3 provides the foundations for the algorithm presented in section 3.4.

3.1 Definitions

Linear recursive Datalog programs or queries are the focus of this chapter. They consist of one or more nonrecursive exit rules and one linear recursive rule. Throughout this paper we will assume that there is only one exit rule per linear recursive query. In the case that there are several exit rules they can be collapsed to one single exit rule:

$$\begin{aligned} p(\ldots) \quad &:- \quad e_1(\ldots). \\ &\ldots \\ p(\ldots) \quad &:- \quad e_k(\ldots). \end{aligned}$$

can be replaced by

$$\begin{aligned} e'(\ldots) \quad &:- \quad e_1(\ldots). \\ &\ldots \\ e'(\ldots) \quad &:- \quad e_k(\ldots). \end{aligned}$$

$$p(\ldots) \quad :- \quad e'(\ldots).$$

where e' is a new predicate symbol. In terms of relational algebra, E' (this is the relation corresponding to the predicate e') represents the union of all E_i's (these are the relations corresponding to the e_i predicates). So the class of of linear recursive queries is defined as follows.

Definition 1 (Linear Recursive Query): A *linear recursive query* consists of a linear recursive rule and at least one exit rule. They look like this:

$$
\begin{array}{lll}
p(X,Z) \; :- \; e(X,Z). & & \text{Exit Rule} \\
p(X,Z) \; :- \; p(X,Y), & q_1(X_1,Y_1,Z_1,W_1), & \text{Linear Recursive Rule} \\
& q_2(X_2,Y_2,Z_2,W_2), & \\
& \ldots & \\
& q_n(X_n,Y_n,Z_n,W_n). &
\end{array}
$$

with the restrictions:

- $n \geq 1$ (i. e. there is at least one q_i-literal)
- for $1 \leq i \leq n$
  - q_i has a corresponding database relation Q_i that has already been computed (if Q_i is an IDB relation or is stored somewhere in the database (if Q_i is an EDB relation
  - $X, Y, Z, X_i, Y_i, Z_i, W_i$ are vectors[1] of variables or constants
  - $X_i \subseteq X, Y_i \subseteq Y, Z_i \subseteq Z$
- $$Z = \bigcup_{i=1}^{n} Z_i$$

  i. e. the linear recursive rule is *safe*.
- X, Y, Z and
  $$W := \bigcup_{i=1}^{n} W_i$$
  are pairwise disjoint, i. e. $X \cap Y = \emptyset, X \cap Z = \emptyset, X \cap W = \emptyset, Y \cap Z = \emptyset, Y \cap W = \emptyset, Z \cap W = \emptyset$
- For the predicate e of the exit rule there is a corresponding relation E that is already known. E is called the *initial relation* of the relation P.

[1] in the sense of ordered lists. Sometimes the order of the list elements is of no importance. In this case set operators like $\subseteq$, $\cup$, etc. are applied to these lists.

- The literal $p(X,Z)$ in the head of the linear recursive rule is called the *consequent* of the rule whereas $p(X,Y)$ of the body is called the *antecedent*.

Starting with the tuples of E and applying the linear recursive rule repeatedly (i. e. doing a bottom-up evaluation) we get in each step a new *generation* of tuples. So E is called the generation 0. Tuples that are computed by applying the recursive rule once to tuples of generation k are said to be of generation $k+1$. The generations of P are denoted by $P_0, P_1, P_2, \ldots$

Let r denote a linear recursive rule. $r(T)$ denotes the result of an application of r to an arbitrary relation T: if

$$P(\ldots) = P(\ldots) \bowtie Q(\ldots)$$

is the recursive Datalog equation corresponding to r then T is used to eliminate recursion so

$$r(T) = T(\ldots) \bowtie Q(\ldots)$$

So the bottom-up evaluation of a linear recursive rule r and its exit relation E can be described as computing $P_1 = r(E)$, $P_2 = r(r(E))$, $P_3 = r(r(r(E)))$, ...

r^j denotes the *j-step rule* of r which has the same effect on an arbitrary relation T as applying r j times to T, i. e. :

$$r^j(T) = \underbrace{r \circ r \circ \ldots \circ r}_{j}(T) = \underbrace{r(r(\ldots r}_{j}(T)\ldots))$$

So this implies:

$$P_{k+j} = r^j(P_k)$$

3.2 Evaluation by Using Transitive Closure Operations

We will follow the approach of [Jag 87] because of its usefulness for a wide range of queries.

As seen by examples 4 (see page 14) and 5 (page 15) some linear recursive queries can be easily transformed to an expression of relational algebra possibly involving an additional operator for transitive closure. In general this can be done for linear recursive queries that are in the *basic form for transitive closure* [Jag 87]:

$$p(X,Z)\ :-\ p(X,Y),\ q(Y,Z). \tag{3.1}$$

where X,Y and Z are pairwise disjoint vectors of variables. Therefore the goal is to derive a rule in the basic form, obtained from an arbitrary linear recursive rule, which can be used to process the query. Although [Jag 87] claims that this can be done for *every* linear recursive rule, problems arise for queries like the one in example 5: there is a common variable of the two

p-literals and the q-literal. Such rules cannot be further transformed to the basic form so we will use the *general basic form for transitive closure*

$$p(X,Z) \ :- \ p(X,Y),\ q(X',Y,Z). \tag{3.2}$$

where X,Y,Z are pairwise disjoint sets of variables and $X' \subseteq X$. [Jag 87] is right in the sense that (3.2) can be evaluated by a *data-dependent* number of rules of form (3.1): let $\{x_1, \ldots, x_n\}$ be the possible values for the X' attributes. Then the relation Q can be split into n partitions $Q_1, \ldots, Q_n$ by using the n rules

$$\begin{aligned} q_1(Y,Z) \quad &:- \quad q(x_1,Y,Z). \\ &\ldots \\ q_n(Y,Z) \quad &:- \quad q(x_n,Y,Z). \end{aligned}$$

so n rules of the basic form (3.1) can be used to process the general basic form (3.2):

$$\begin{aligned} p(Y,Z) \quad &:- \quad p(X,Y),\ q_1(Y,Z). \\ &\ldots \\ p(Y,Z) \quad &:- \quad p(X,Y),\ q_n(Y,Z). \end{aligned}$$

In the next section, the techniques are described which are used to derive automatically a rule in the general basic form for transitive closure such that this rule can be used to process a given arbitrary linear recursive rule r.

3.3 Isomorphism and Substitution Graphs

The most important point of the algorithm presented by [Jag 87] is that it rewrites a linear recursive rule to a *canonical form for transitive closure*:

$$p(A) \ :- \ p(B),\ q_1(C_1),\ \ldots\ q_m(C_m). \tag{3.3}$$

where the argument vectors A and B are *isomorphic* in a sense to be defined below. Expression (3.3) can, in turn, be reduced to the general basic form for transitive closure[2]

$$p(X,Z) \ :- \ p(X,Y),\ q(X',Y,Z).$$

[2] See page 21.

where $X' \subseteq X$. Because A and B of (3.3) are isomorphic A can be split to (X, Z) and B to (X, Y) where X is the vector of variables that are common to A and B and Z respectively Y are the 'private' parts of A respectively B.

To analyze a linear recursive rule especially the point when isomorphism is met [Jag 87] uses substitution graphs. Once these constructs have been defined a theorem provides the foundation for the efficient implementation of the algorithm.

3.3.1 Definitions

Definition 2 (Substitution Graph): A *substitution graph* for a linear recursive rule

$$p(X_1, \ldots, X_n) \ :- \ p(Y_1, \ldots, Y_n),\ q(\ldots).$$

where all X_i, Y_i are variables or constants, is defined thus:

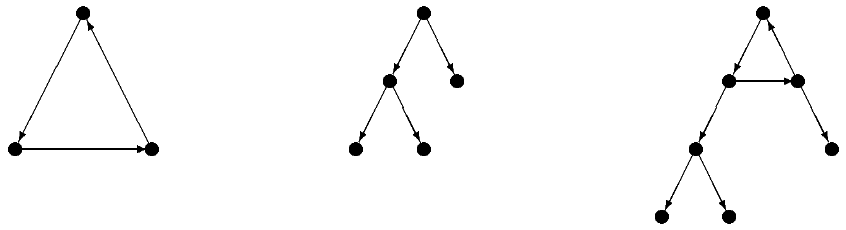
1. There is a vertex for each argument position i $(1 \leq i \leq n)$.
2. There is a directed (!) edge from vertex i to vertex j if $Y_j = X_i$, i.e. a variable or constant (say W) is in the head of the rule at position i (i.e. $X_i = W$) and occurs in position j of the recursive literal in the body of the rule (i.e. $Y_j = W$).

It is possible that some vertices are not connected to any edges. This implies that either the consequent holds a variable at this position that is not used by the antecedent of the rule (so no edge is leaving this vertex). or there is no predecessor for the vertex because the antecedent contains a new argument at this position. A substitution graph is a special kind of graph as it consists only of disconnected components each of which is a cycle, a tree or a cycle with one or more trees 'hanging off' the vertices of the cycle. This is because a vertex of the graph has at most one predecessor. Actually it represents a substitution.

Definition 3 (Substitution): A *substitution* S is a mapping of an argument vector $X = (X_1, \ldots, X_n)$ to an argument vector $Y = (Y_1, \ldots, Y_n)$; more briefly $Y = S(X)$ where Y denotes the argument vector of the antecedent and X that of the consequent of a linear recursive rule. $S^k(X)$ denotes k successive applications of S to X:

$$S^k(X) = \underbrace{S(S(\ldots S}_{k}(X)\ldots))$$

Note that the X_i and Y_i are arguments, i. e. variables *or* constants. If Y contains a variable Y_i that does not appear in X and a substitution S is defined by $Y = S(X)$ then this means that in every application of S to an arbitrary argument vector X' the result $Y' = S(X')$ will hold a



(a) (b) (c)

Figure 3.1: Types of connected components in substitution graphs

new variable on the position that corresponds to that of Y_i in Y. For example if $X = (A, B)$ and $Y = (B, C)$ and $S(X) = Y$ then

$$S((G, H)) = (H, I) \tag{3.4}$$

$$S((a, H)) = (H, I) \tag{3.5}$$

Definition 4 (Isomorphism): Two argument vectors $X = (x_1, \ldots, x_m)$ and $Y = (y_1, \ldots, y_m)$ are *isomorphic* if

- $x_i = x_j \quad \Leftrightarrow \quad y_i = y_j$
- $x_i = y_i$ or x_i does not appear in Y (and vice versa)
- any non-common arguments are variables

For example (U, a, U, b, a, Y) and (X, a, X, b, a, Y) are isomorphic whereas (S, T, a) and (X, Y, b) are not (because of the third argument position).

3.3.2 Usefulness of Isomorphism and Substitutions Graphs

A connected component of a substitution graphs can be one of three types of directed graph structure (see figure 3.1): a cycle (a), a tree (b) and a combination of one cycle with one or more trees hanging from this cycle (c). The length of a cycle is the number of edges involved in the cycle. The depth of a tree is the maximum length of a path from the root to a leaf, where the length of a path is the number of edges involved in the path. For example: the cycle (a) of figure 3.1 is of length 3; the tree (b) is of depth 2; the cycle involved in (c) is of length 3 and there are two trees, one of depth 2 and one of depth 1.

A substitution graph shows how variable names on the argument positions are shifted if the linear recursive rule is applied to itself: for example the linear recursive rule r

$$p(A, B, C) \;:-\; p(C, A, B),\; q(A). \tag{3.6}$$

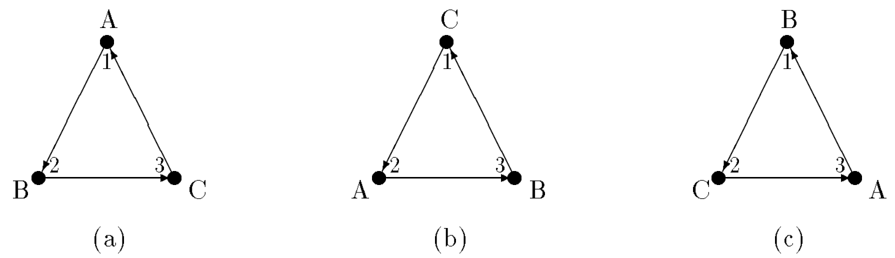


Figure 3.2: Behaviour of a substitution

has the substitution graph of figure 3.2. If it is applied to itself – i. e. the antecedent is extended by applying r – the result is

$$p(A,B,C)\ :-\ p(B,C,A),\ q(C),\ q(A).$$

which we refer to as the 2-step-rule, denoted r^2, of r because it has the same effect as applying r two times consecutively. Finally the 3-step-rule r^3 is

$$p(A,B,C)\ :-\ p(A,B,C),\ q(B),\ q(C),\ q(A).$$

The interesting fact about r^3 is that the argument vectors of the p-literals are isomorphic, in this case even identical.

The relationship between substitution graphs, substitutions and j-step rules is the following: a substitution graph is the graphic representation of a substitution. For example the substitution S corresponding to the rule (3.6) is

$$S\begin{pmatrix} A \\ B \\ C \end{pmatrix} = \begin{pmatrix} C \\ A \\ B \end{pmatrix} \tag{3.7}$$

which says that the first argument A of (A,B,C) is shifted to the second position, the second argument B to the third and the third argument C to the first.

The same behaviour can be seen in the substitution graph (figure 3.2): the vertices correspond to the argument positions and the labels to the variable names on these positions. Figure 3.2 (b) shows the result of one iteration i. e. one application of the substitution. It holds exactly the argument vector $(C,A,B) = S(A,B,C)$ of the antecedent of the 1-step-rule r. The figure shows also the result of another iteration which corresponds to $(B,C,A) = S(C,A,B) = S^2(A,B,C)$ – the argument vector of the antecedent in the 2-step-rule r^2 of the example.

Substitution graphs are very useful in the sense that it is easier to show certain properties of the underlying substitutions. The interesting question for the algorithm of [Jag 87] is when isomorphism is met. The answer will be given by the following lemmas and the final theorem.

Lemma 1 *Let G be the substitution graph corresponding to a substitution S of a linear recursive rule*

$$p(X)\ :-\ p(Y),\ \ldots$$

where X and Y are argument vectors and $S(X) = Y$. If G consists only of a cycle with possibly some trees hanging from this cycle and

d = *length of the cycle of G*

g = *maximum depth of all trees attached to the cycle of G*

then the following holds:

$S^k(X)$ *and* $S^{k+jd}(X)$ *are isomorphic for $k \geq g$ and $j \geq 0$.*

Proof

1. Trivially each vertex of the cycle holds the same label every d-th iteration of the mapping (see figure 3.3 for an example).

2. As the roots of all trees are involved in this cycle all trees are fed by labels of the cycle vertices (see iterations 1 - 5 of figure 3.3).

3. Consequently G only holds labels that originated in the cycle vertices after g generations; all original labels of the tree vertices are pushed out (see iterations 2 - 5 of figure 3.3). So $S^k(X)$ and $S^l(X)$ hold only common variables for $k, l \geq g$.

4. Repetition of labels is only possible in the tree vertices[3]. The pattern of repetition of labels is established after g iterations because the tree-root-labels have reached or passed all the leaves. This is only of importance if there is at least one tree involved. If there is no tree (i. e. $g = 0$) then there is no repetition of variables. So $S^k(X)$ and $S^l(X)$ hold the same pattern of repetition for $k, l \geq g$.

The conclusion of 1., 2., 3. and 4. is that $S^k(X)$ and $S^{k+jd}(X)$ are identical for $k \geq g$ and $j \geq 0$ so the statement of the lemma holds. □

Example 7: Figure 3.4 gives an example of a substitution graph of 12 vertices. Numbers are used as labels. In the beginning the vertices are labeled by their respective position. The table shows the results of some iterations. The leftmost column holds the number of the iteration. Note that from iteration 3 onwards the vertices hold only labels '1', '2' and '3'.

[3]These are the only vertices that have more than one successor. Repetition of labels in the cycles is no problem as the pattern of repetition is established by the initialization. Apart from that the optimization algorithm avoids repeated labels in the initial state as we will see later.

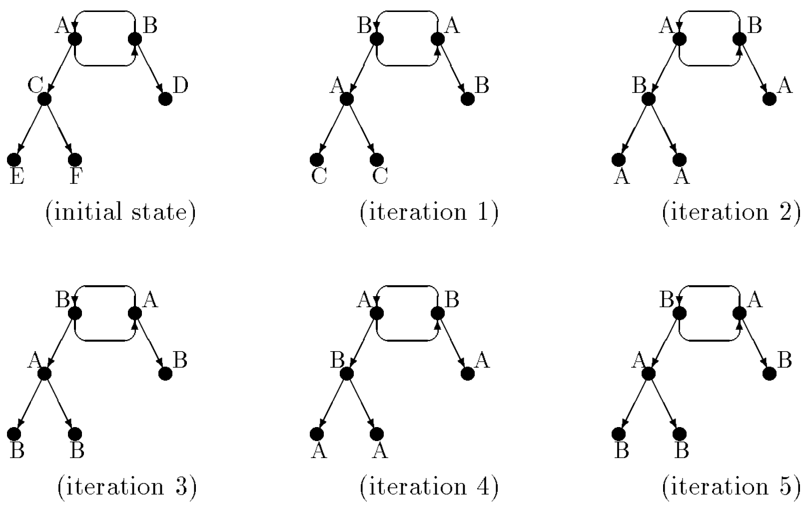


Figure 3.3: Moving labels in a cycle with trees 'hanging off'

These are the labels that originated in the cycle vertices. It is easy to see that $d = 3$ and $g = 3$ so we know from the lemma that $S^k(X)$ and $S^{k+3}(X)$ are isomorphic for $k \geq 3$. This fact can be seen in the table of figure 3.4.

Lemma 2 *Let G be the substitution graph of a substitution S of a linear recursive rule*

$$p(X) \; :- \; p(Y), \; \ldots$$

where X and Y are argument vectors and $S(X) = Y$. If G consists of only a tree and

d = *depth of the tree + 1*

g = *depth of the tree*

then the following holds:

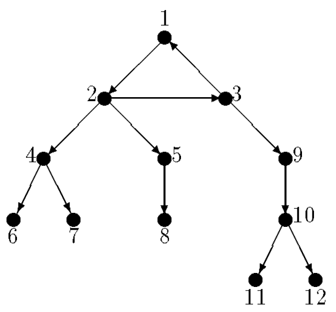
$S^k(X)$ *and* $S^{k+j}(X)$ *are isomorphic for* $k \geq g$ *and* $j \geq d$.

Proof

1. After g iterations the pattern of repetition is established because the root label has reached or passed all the leaves so $S^k(X)$ and $S^l(X)$ hold the same pattern of repetition for $k, l \geq g$. For an example see figure 3.6.

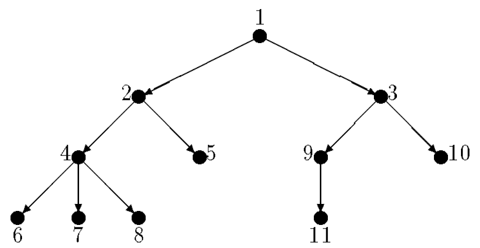
2. After d iterations all original labels are shifted out so $S^k(X)$ and $S^{k+j}(X)$ have no common variable names for $k \geq 0$ and $j \geq d$.

The conclusion of 1. and 2. is the statement of the lemma. $\square$



0.	1	2	3	4	5	6	7	8	9	10	11	12
1.	3	1	2	2	2	4	4	5	3	9	10	10
2.	2	3	1	1	1	2	2	2	2	3	9	9
3.	1	2	3	3	3	1	1	1	1	2	3	3
4.	3	1	2	2	2	3	3	3	3	1	2	2
5.	2	3	1	1	1	2	2	2	2	3	1	1
6.	1	2	3	3	3	1	1	1	1	2	3	3
7.	3	1	2	2	2	3	3	3	3	1	2	2
8.	2	3	1	1	1	2	2	2	2	3	1	1
9.	1	2	3	3	3	1	1	1	1	2	3	3
10.	3	1	2	2	2	3	3	3	3	1	2	2
11.	2	3	1	1	1	2	2	2	2	3	1	1
12.	1	2	3	3	3	1	1	1	1	2	3	3

Figure 3.4: A substitution graph with $d = 3$ and $g = 3$.



0.	1	2	3	4	5	6	7	8	9	10	11
1.	12	1	1	2	2	4	4	4	3	3	9
2.	13	12	12	1	1	2	2	2	1	1	3
3.	14	13	13	12	12	1	1	1	12	12	1
4.	15	14	14	13	13	12	12	12	13	13	12
5.	16	15	15	14	14	13	13	13	14	14	13
6.	17	16	16	15	15	14	14	14	15	15	14
7.	18	17	17	16	16	15	15	15	16	16	15
8.	19	18	18	17	17	16	16	16	17	17	16
9.	20	19	19	18	18	17	17	17	18	18	17
10.	21	20	20	19	19	18	18	18	19	19	18
11.	22	21	21	20	20	19	19	19	20	20	19
12.	23	22	22	21	21	20	20	20	21	21	20

Figure 3.5: A substitution graph with $d = 4$ and $g = 3$.

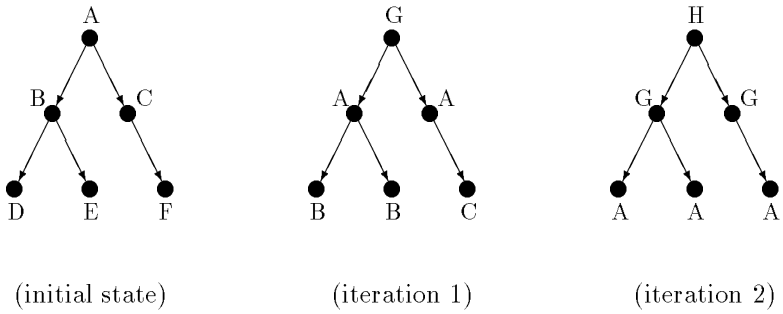


Figure 3.6: Pattern of repetition in a tree

Example 8: Figure 3.5 gives an example of a substitution graph of 11 vertices. It is easy to see that $d = 4$ and $g = 3$ so we know from the lemma that $S^k(X)$ and $S^{k+4}(X)$ are isomorphic for $k \geq 3$. This example shows the reason why d is defined to be the depth of the tree **+ 1** in this case. [Jag 87]'s general definition of d results in $d = 3$ which is obviously wrong because e. g. $S^5(X)$ and $S^8(X)$ are not isomorphic (see table): the label '16' appears in $S^5(X)$ on position 1 and in $S^8(X)$ on positions 6, 7, 8 and 11. So $S^5(X)$ and $S^8(X)$ are not isomorphic.

In the following theorem two substitutions S_1 and S_2 will be combined to one substitution S:

$$S(X) = S((X_1, X_2)) = (S_1(X_1),\, S_2(X_2))$$

which means that the argument vector X consists of two parts X_1 and X_2. Applying S to X means applying S_1 to X_1 and S_2 to X_2 and concatenating the results $S_1(X_1)$ and $S_2(X_2)$ to get the final result for $S(X)$. This is no problem (regarding interferences between the domains of S_1 and S_2) as in the definition of a substitution argument names are only used to describe to which position they are transferred to. See (3.4) and (3.7).

Theorem 1 *Let G be a substitution graph (corresponding to a substitution S) consisting of two disjoint subgraphs G_1 and G_2 with the parameters d_1, g_1, d_2, g_2 and the corresponding substitutions S_1, S_2 respectively. Let $X = (X_1, X_2)$ and $Y = (Y_1, Y_2)$ be argument vectors and X_i, Y_i be the parts of X and Y that represent the argument positions of G_i ($i \in \{1, 2\}$).*

a.) If G_1 contains a cycle and G_2 does not then let

$$d = \text{the least multiple of } d_1 \text{ that is larger than } d_2$$

b.) If G_1 does not contain a cycle and G_2 does then let

$$d = \text{the least multiple of } d_2 \text{ that is larger than } d_1$$

c.) If G_1 and G_2 both contain cycles then let

$$d = \text{the least common multiple of } d_1 \text{ and } d_2$$

d.) If G_1 and G_2 are bot acyclic then let

$$d = max(d_1, d_2)$$

and let $g = max(g_1, g_2)$, then the following holds

$$S^k(X) \text{ and } S^{k+jd}(X) \text{ are isomorphic for } k \geq g \text{ and } j \geq 0.$$

Proof

- We note that $S^k(X)$ for $k \geq 0$ can be derived by

$$\begin{aligned}
S^k(X) &= S^k((X_1, X_2)) \\
&= S^{k-1}(S((X_1, X_2))) \\
&= S^{k-1}((S_1(X_1), S_2(X_2))) \\
&= \cdots \\
&= (S_1^k(X_1),\ S_2^k(X_2)) \\
\text{so} \quad S^{k+jd}(X) &= (S_1^{k+jd}(X_1),\ S_2^{k+jd}(X_2))
\end{aligned}$$

- Without losing generality G_2 is assumed to consist of only one connected component.
- The proof will be an induction over c where

$$c = \text{number of connected components in } G$$

Base Case: $c = 2$, i. e. G_1 consists of one connected component.

a.) G_1 contains a cycle, G_2 is a tree:

By lemma 1 $S_1^k(X_1)$ and $S_1^{k+j\,d}(X_1)$ are

isomorphic for $k \geq \max(g_1, g_2) \geq g_1$ because d is a multiple of d_1.

By lemma 2 $S_2^k(X_2)$ and $S_2^{k+j\,d}(X_2)$ are

isomorphic for $k \geq \max(g_1, g_2) \geq g_2$ because $d \geq d_2$.

So the respective concatenations $(S_1^k(X_1),\, S_2^k(X_2)) = S^k(X)$ and $(S_1^{k+j\,d}(X_1),\, S_2^{k+j\,d}(X_2)) = S^{k+j\,d}(X)$ are isomorphic.

b.) G_1 is a tree, G_2 contains a cycle:

This case is similar to a.) – the roles of G_1 and G_2 are just swapped.

c.) G_1 contains a cycle, G_2 contains a cycle:

By lemma 1 $S_1^k(X_1)$ and $S_1^{k+j\,d}(X_1)$ are

isomorphic for $k \geq \max(g_1, g_2) \geq g_1$ because d is a multiple of d_1.

By lemma 1 $S_2^k(X_2)$ and $S_2^{k+j\,d}(X_2)$ are

isomorphic for $k \geq \max(g_1, g_2) \geq g_2$ because d is a multiple of d_2.

So the respective concatenations $(S_1^k(X_1),\, S_2^k(X_2)) = S^k(X)$ and $(S_1^{k+j\,d}(X_1),\, S_2^{k+j\,d}(X_2)) = S^{k+j\,d}(X)$ are isomorphic.

d.) G_1 is a tree, G_2 is a tree:

By lemma 2 $S_1^k(X_1)$ and $S_1^{k+j\,d}(X_1)$ are

isomorphic for $k \geq \max(g_1, g_2) \geq g_1$ because $d \geq d_1$.

By lemma 2 $S_2^k(X_2)$ and $S_2^{k+j\,d}(X_2)$ are

isomorphic for $k \geq \max(g_1, g_2) \geq g_2$ because $d \geq d_2$.

So the respective concatenations $(S_1^k(X_1),\, S_2^k(X_2)) = S^k(X)$ and $(S_1^{k+j\,d}(X_1),\, S_2^{k+j\,d}(X_2)) = S^{k+j\,d}(X)$ are isomorphic.

Assumption: The theorem holds for all G with $c \leq c_0$. $(*)$

Inductive Step: $c = c_0 + 1$, i. e. G_1 has c_0 connected components

$\Rightarrow$ because of $(*)$ the theorem holds for G_1 $(**)$

a.) G_1 contains a cycle, G_2 is a tree:

$S_1^k(X_1)$ and $S_1^{k+j\,d}(X_1)$ are isomorphic for $k \geq \max(g_1, g_2) \geq g_1$

because of $(**)$ and the fact that d is a multiple of d_1.

By lemma 2 $S_2^k(X_2)$ and $S_2^{k+j\,d}(X_2)$ are

isomorphic for $k \geq \max(g_1, g_2) \geq g_2$ because $d \geq d_2$.

So the respective concatenations $(S_1^k(X_1),\, S_2^k(X_2)) = S^k(X)$ and $(S_1^{k+j\,d}(X_1),\, S_2^{k+j\,d}(X_2)) = S^{k+j\,d}(X)$ are isomorphic.

b.) G_1 consists of trees, G_2 contains a cycle:

This case is similar to a.)

c.) G_1 contains a cycle, G_2 contains a cycle:

$S_1^k(X_1)$ and $S_1^{k+jd}(X_1)$ are isomorphic for $k \geq \max(g_1, g_2) \geq g_1$
because of (**) and the fact that d is a multiple of d_1.
By lemma 1 $S_2^k(X_2)$ and $S_2^{k+jd}(X_2)$ are
isomorphic for $k \geq \max(g_1, g_2) \geq g_2$ because d is a multiple of d_2.
So the respective concatenations $(S_1^k(X_1),\, S_2^k(X_2)) = S^k(X)$ and
$(S_1^{k+jd}(X_1),\, S_2^{k+jd}(X_2)) = S^{k+jd}(X)$ are isomorphic.

d.) G_1 is a tree, G_2 is a tree:
$S_1^k(X_1)$ and $S_1^{k+jd}(X_1)$ are isomorphic for $k \geq \max(g_1, g_2) \geq g_1$
because of (**) and the fact that $d \geq d_1$.
By lemma 2 $S_2^k(X_2)$ and $S_2^{k+jd}(X_2)$ are
isomorphic for $k \geq \max(g_1, g_2) \geq g_2$ because $d \geq d_2$.
So the respective concatenations $(S_1^k(X_1),\, S_2^k(X_2)) = S^k(X)$ and
$(S_1^{k+jd}(X_1),\, S_2^{k+jd}(X_2)) = S^{k+jd}(X)$ are isomorphic.

Therefore the statement of the theorem holds. $\square$

d is also said to be the *diameter* of G. The advantage of the two lemmas and the theorem is that they give smaller values for d and g than the theorem of [Jag 87]. This reduces the number of (computationally expensive) joins.

3.4 The Algorithm

In section 3.4.1 we will summarize the algorithm described in section 6 of [Jag 87]. After that we will give an alternative view at the algorithm by arranging the generations produced by a linear recursive rule r in a matrix-form.

3.4.1 Stage-by-Stage

In general the algorithm starts with a query that has the form:

$$
\begin{array}{llr}
p(X,Z) :- e(X,Z). & & \text{Exit Rule} \\
p(X,Z) :- p(X,Y), & q_1(X_1,Y_1,Z_1,W_1), & \text{Linear Recursive Rule} \\
 & q_2(X_2,Y_2,Z_2,W_2), & \\
 & \ldots & \\
 & q_n(X_n,Y_n,Z_n,W_n). &
\end{array}
$$

X, Y, Z and X_i, Y_i, Z_i, W_i are vectors of variables. For further restrictions see section 3.1.

Actually the algorithm's emphasis is on the linear recursive rule r that will be transformed to the general basic form for transitive closure (see (3.2) on page 21):

$$p(X,Z) \ :- \ p(X,Y), q(X',Y,Z). \tag{3.8}$$

where $p(X,Z)$ contains no repeated arguments and the vectors (X,Z) and (X,Y) are isomorphic.

1. **Project out W_i:** For each literal $q_i(X_i,Y_i,Z_i,W_i)$ define a (nonrecursive) rule

$$q_i'(X_i,Y_i,Z_i) \ :- \ q_i(X_i,Y_i,Z_i,W_i).$$

   r looks now like that:

$$\begin{aligned} p(X,Z) \quad :- \quad & p(X,Y), \\ & q_1'(X_1,Y_1,Z_1), \\ & \ldots \\ & q_n'(X_n,Y_n,Z_n). \end{aligned}$$

2. **Summarize q_i':** Define a new (nonrecursive) rule

$$q'(X',Y,Z) \ :- q_1'(X_1,Y_1,Z_1), \ldots, q_n'(X_n,Y_n,Z_n).$$

   to summarize all the q_i'. Remember that $X' := \bigcup_{i=1}^{n} X_i$, $Y := \bigcup_{i=1}^{n} Y_i$, $Z := \bigcup_{i=1}^{n} Z_i$ and $X' \subseteq X$ (see section 3.1). r has now the form:

$$p(X,Z) \ :- \ p(X,Y), q'(X',Y,Z).$$

   We have nearly reached the form of (3.2) but it is necessary to eliminate repeated arguments in the consequent $p(X,Z)$ (if there are any) and – furthermore – (X,Z) and (X,Y) may not be isomorphic.

3. **Eliminate repeated arguments in the consequent $p(X,Z)$:** If there are no repeated arguments skip this stage. Otherwise compute the *fan-in f* of the graph as described by [Jag 87]. It uses the substitution graph with *reversed* edges – see [Jag 87] for details :

   - Each vertex holds a label with the number of its corresponding argument position.
   - Collapse any vertices without predecessors if the corresponding positions in the consequent contain the same argument. Also collapse vertices having the same predecessor. The resulting new vertices are labeled with the subset of labels of the original vertices.

- Collapse vertices having the same predecessor. Repeat this step until there are no changes in the graph.

$$f = \text{number of iterations of this last step} + 1$$

Compute the generations $P_0, P_1, \ldots, P_{f-1}, P_f$ and store

$$P' = P_0 \cup P_1 \cup \ldots \cup P_{f-1}$$

in the database. Derive a new recursive rule r from the old one by using the argument pattern that stabilizes from generation P_f on. Positions that will hold the same values are those that appear in the same vertex of the resulting graph of the 'fan-in-computation'. Leave out the repeated arguments. This has to be done for the recursive literal in the head *and* in the body of the rule. Use P_f as the new initial relation P_0.

4. **Compute the parameters d and g of the substitution graph:** Compute the substitution graph of the linear recursive rule. See section 3.3 for lemmas 1 and 2 and theorem 1 to give a description of how to compute d and g.

5. **Compute the first g generations:** Starting with the initial relation $P_0 = E$ (see exit rule – only if stage 3 is not relevant; otherwise use the new P_0 computed in stage 3) compute the generations $P_1, P_2, \ldots, P_g$ by a bottom-up application of r.

6. **Compute the d-step-rule r^d:** Compute the d-step rule r^d of r by applying r $(d-1)$ times to itself. Now use $S^g((X,Z))$ as the argument vector of the consequent and $S^{g+d}((X,Z))$ as the argument vector of the antecedent. So r^d looks like this:

$$\begin{aligned} p(S^g((X,Z))) \quad :- \quad & p(S^{g+d}((X,Z))), \\ & q'(X',Y,Y_2),\, q'(X',Y_2,Y_3),\, \ldots,\, q'(X',Y_d,Z). \end{aligned}$$

Finally we can use a (nonrecursive) rule to summarize the sequence of q' literals. Note that $Y_2, Y_3, \ldots, Y_d$ are variables that are only used inside, i. e. they do not affect literals outside the sequence:

$$q(X',Y,Z) \ :- \ q'(X',Y,Y_2),\, q'(X',Y_2,Y_3),\, \ldots,\, q'(X',Y_d,Z).$$

So the d-step rule r^d looks more briefly:

$$p(S^g((X,Z))) \ :- \ p(S^{g+d}((X,Z))), q(X',Y,Z).$$

7. **Eliminate repeated arguments in the consequent:** This is no problem because $S^g((X,Z))$ and $S^{g+d}((X,Z))$ are isomorphic so repeated arguments can be omitted. Finally r^d has the following form with $A' \subseteq A \subseteq X, B \subseteq Y, C \subseteq Z, \ldots$, (A,C) and (A,B) are isomorphic and (A,C) has no repeated arguments, so r^d is of form (3.2):

$$p(A,C) \; :- \; p(A,B),\; q(A',B,C).$$

8. **Use a transitive closure algorithm:** Now we can use a simple transitive closure algorithm to compute a number of transitive closures – one for each partition Q_i[4] of the relation Q – because the d-step rule r^d is in the appropriate form. Instead of using E as the initial relation we will use P_g. Q is the corresponding relation of the predicate q. So compute the relation R:

$$\begin{aligned} R &= P_g \,\cup\, r^d(P_g) \,\cup\, r^{2d}(P_g) \,\cup\, \ldots \\ &= P_g \,\cup\, P_{g+d} \,\cup\, P_{g+2d} \,\cup\, \ldots \\ &= P_g \bowtie \underbrace{\bigcup_{i=1}^{m} \left[\{x_i\} \times Q_i^*\right]}_{=:\, Q^{**}} \end{aligned}$$

9. **Replicate eliminated arguments:** In stage 7 some arguments have been eliminated. Therefore we have to replicate them by duplicating some columns of the relation R.

10. **Filling the gap:** Up to now we have got the generations $P_0, P_1, P_2, \ldots, P_g$ (see stage 5) and $P_g, P_{g+d}, P_{g+2d}, \ldots$ (see stage 8). There are gaps of $(d-1)$ generations between P_{g+kd} and $P_{g+(k+1)d}$ where $k \geq 0$. We can fill this gaps by applying the first $(d-1)$ j-step rules to R or by applying r (d-1) times to R:

$$\begin{aligned} r(R) &= P_{g+1} \,\cup\, P_{g+d+1} \,\cup\, P_{g+2d+1} \,\cup\, \ldots \\ r^2(R) &= P_{g+2} \,\cup\, P_{g+d+2} \,\cup\, P_{g+2d+2} \,\cup\, \ldots \\ &\ldots \\ r^{d-1}(R) &= P_{g+d-1} \,\cup\, P_{g+2d-1} \,\cup\, P_{g+3d-1} \,\cup\, \ldots \end{aligned}$$

Finally all generations are available from stages 5, 8 and 10:

$$P = \underbrace{P_0 \cup P_1 \cup \ldots \cup P_{g-1}}_{5} \,\cup\, \underbrace{R}_{8} \,\cup\, \underbrace{r(R) \,\cup\, r^2(R) \,\cup\, \ldots \,\cup\, r^{d-1}(R)}_{10} \tag{3.9}$$

[4] Remember that Q_i is in fact a selection of the tuples of a relation Q with attribute sets (X,Y,Z). All tuples of Q_i are generated from tuples of Q with the same value in the X attributes and the X attributes are projected out: $Q_i = \pi_{Y,Z}(\sigma_{X=x_i}(Q))$.

11. **Replicate some eliminated arguments:** In stage 3 some repeated arguments had been eliminated. Therefore we have to duplicate certain columns of P to replicate these values. Finally add the tuples of P' (see stage 3) to P.

3.4.2 An Example for the Algorithm

The resulting generations $P_0, P_1, P_2, \ldots$ can be ordered in a $t \times d$ matrix where d is the diameter of the substitution graph and t is one more than the number of the applications of the d-step rule r^d that are necessary to compute the closure (see stage 8 of section 3.4.1):

$$\begin{aligned} R &= P_g \cup r^d(P_g) \cup r^{2d}(P_g) \cup \ldots \\ &= P_g \cup r^d(P_g) \cup r^{2d}(P_g) \cup \ldots \cup r^{(t-1)d}(P_g) \end{aligned}$$

because $r^{td}(P_g) = r^{(t+1)d}(P_g) = \ldots = \emptyset$. The proof of this property is given by [Jag 87]. The idea of the matrix will be introduced by an example:

Example 9: The substitution graph of this query is very simple: there is only one cycle of length 4, so $d = 4$ and $g = 0$:

$$\begin{aligned} p(A, B, C, D, Z) &:- \quad e(A, B, C, D, Z). \\ p(A, B, C, D, Z) &:- \quad p(D, A, B, C, Y),\ q(Y, Z). \end{aligned}$$

with the EDB relations E and Q:

$$\begin{aligned} E &= \{(0,1,1,0,2), (1,0,1,0,3), (1,1,0,0,2), (0,0,1,1,2)\} \\ Q &= \{(1,2), (2,3), (3,4), (4,5),\ \ldots\ , (16,17)\} \end{aligned}$$

Let us apply the algorithm of section 3.4.1: It is not necessary to execute stages 1 to 3 because the linear recursive rule is already in the appropriate form. Stage 4 results in $d = 4$ and $g = 0$ as mentioned above. Because of $g = 0$ and $P_0 = E$ already exists stage 5 can be skipped. The next task is to compute the first 4 'step-rules' (stage 6):

$$\begin{aligned} p(A, B, C, D, Z) &:- \quad p(D, A, B, C, Y_1),\ q(Y_1, Z). \\ p(A, B, C, D, Z) &:- \quad p(C, D, A, B, Y_2),\ q(Y_2, Y_1),\ q(Y_1, Z). \\ p(A, B, C, D, Z) &:- \quad p(B, C, D, A, Y_3),\ q(Y_3, Y_2),\ q(Y_2, Y_1),\ q(Y_1, Z). \\ p(A, B, C, D, Z) &:- \quad p(A, B, C, D, Y_4),\ q(Y_4, Y_3),\ q(Y_3, Y_2),\ q(Y_2, Y_1),\ q(Y_1, Z). \end{aligned}$$

The last rule (the 4-step rule r^4) can be reduced by using a new rule for qq or – in terms of relations – by using a new relation QQ:

$$\begin{aligned} qq(Y,Z) \quad &:- \quad q(Y,Y_3),\ q(Y_3,Y_2),\ q(Y_2,Y_1),\ q(Y_1,Z). \\ p(A,B,C,D,Z) \quad &:- \quad p(A,B,C,D,Y),\ qq(Y,Z). \end{aligned}$$

Finally the relation QQ corresponding to the predicate qq looks like that:

$$QQ = \{(1,5),(2,6),(3,7),(4,8),\ \dots\ ,(13,17)\}$$

Comparing Q and QQ shows why r^4 does in fact 4 steps.

The are no repeated arguments in the consequent of r^4 so stage 7 can be skipped. In stage 8 r^4 is evaluated using an appropriate transitive closure algorithm. Starting with $P_g = P_0 = E$ the first column of generations in figure 3.7 is filled. Stage 9 can be skipped because stage 7 was irrelevant. In stage 10 three ($= d-1$) applications of r to the result of stage 8 are sufficient to fill the rest of the matrix. Note that each application produces the generations of one column. Finally stage 11 can also be skipped because stage 3 was irrelevant.

Please note the following facts:

- The generations $P_{16}, P_{17}, P_{18}, \dots$ are empty.
- Each tuple of one generation produces one tuple of the next generation. In figure 3.7 a tuple in row l of generation P_{k+1} is the result of applying r to the tuple of row l of generation P_k.
- The corresponding tuples of the generations P_k and P_{k+d} (i.e. P_{k+4}) are isomorphic.
- Generation P_{k+d} can be computed by applying the d-step rule r^d to generation P_k. So if we know the first row of the matrix (i.e. generations $P_0, \dots, P_{d-1}$) we know the first element of each column. Therefore each column can be filled by using the d-step rule r^d.

3.4.3 An Alternative View of the Algorithm

The generations P_k can be arranged in a $t \times d$ matrix. d can be derived from the substitution graph, so d is a data-independent parameter; t is approx. the depth of the transitive closure so t depends on the length of the longest path in the input-relation for the transitive closure. Therefore t is a data-dependent parameter and is not known before computation which is not of any relevance. Looking at equation (3.9) the major goal of the algorithm suggested by [Jag 87]

$$P_0: \begin{bmatrix} 0 & 1 & 1 & 0 & 2 \\ 1 & 0 & 1 & 0 & 3 \\ 1 & 1 & 0 & 0 & 2 \\ 0 & 0 & 1 & 1 & 2 \end{bmatrix} \quad P_1: \begin{bmatrix} 1 & 1 & 0 & 0 & 3 \\ 0 & 1 & 0 & 1 & 4 \\ 1 & 0 & 0 & 1 & 3 \\ 0 & 1 & 1 & 0 & 3 \end{bmatrix} \quad P_2: \begin{bmatrix} 1 & 0 & 0 & 1 & 4 \\ 1 & 0 & 1 & 0 & 5 \\ 0 & 0 & 1 & 1 & 4 \\ 1 & 1 & 0 & 0 & 4 \end{bmatrix} \quad P_3: \begin{bmatrix} 0 & 0 & 1 & 1 & 5 \\ 0 & 1 & 0 & 1 & 6 \\ 0 & 1 & 1 & 0 & 5 \\ 1 & 0 & 0 & 1 & 5 \end{bmatrix}$$

$$P_4: \begin{bmatrix} 0 & 1 & 1 & 0 & 6 \\ 1 & 0 & 1 & 0 & 7 \\ 1 & 1 & 0 & 0 & 6 \\ 0 & 0 & 1 & 1 & 6 \end{bmatrix} \quad P_5: \begin{bmatrix} 1 & 1 & 0 & 0 & 7 \\ 0 & 1 & 0 & 1 & 8 \\ 1 & 0 & 0 & 1 & 7 \\ 0 & 1 & 1 & 0 & 7 \end{bmatrix} \quad P_6: \begin{bmatrix} 1 & 0 & 0 & 1 & 8 \\ 1 & 0 & 1 & 0 & 9 \\ 0 & 0 & 1 & 1 & 8 \\ 1 & 1 & 0 & 0 & 8 \end{bmatrix} \quad P_7: \begin{bmatrix} 0 & 0 & 1 & 1 & 9 \\ 0 & 1 & 0 & 1 & 10 \\ 0 & 1 & 1 & 0 & 9 \\ 1 & 0 & 0 & 1 & 9 \end{bmatrix}$$

$$P_8: \begin{bmatrix} 0 & 1 & 1 & 0 & 10 \\ 1 & 0 & 1 & 0 & 11 \\ 1 & 1 & 0 & 0 & 10 \\ 0 & 0 & 1 & 1 & 10 \end{bmatrix} \quad P_9: \begin{bmatrix} 1 & 1 & 0 & 0 & 11 \\ 0 & 1 & 0 & 1 & 12 \\ 1 & 0 & 0 & 1 & 11 \\ 0 & 1 & 1 & 0 & 11 \end{bmatrix} \quad P_{10}: \begin{bmatrix} 1 & 0 & 0 & 1 & 12 \\ 1 & 0 & 1 & 0 & 13 \\ 0 & 0 & 1 & 1 & 12 \\ 1 & 1 & 0 & 0 & 12 \end{bmatrix} \quad P_{11}: \begin{bmatrix} 0 & 0 & 1 & 1 & 13 \\ 0 & 1 & 0 & 1 & 14 \\ 0 & 1 & 1 & 0 & 13 \\ 1 & 0 & 0 & 1 & 13 \end{bmatrix}$$

$$P_{12}: \begin{bmatrix} 0 & 1 & 1 & 0 & 14 \\ 1 & 0 & 1 & 0 & 15 \\ 1 & 1 & 0 & 0 & 14 \\ 0 & 0 & 1 & 1 & 14 \end{bmatrix} \quad P_{13}: \begin{bmatrix} 1 & 1 & 0 & 0 & 15 \\ 0 & 1 & 0 & 1 & 16 \\ 1 & 0 & 0 & 1 & 15 \\ 0 & 1 & 1 & 0 & 15 \end{bmatrix} \quad P_{14}: \begin{bmatrix} 1 & 0 & 0 & 1 & 16 \\ 1 & 0 & 1 & 0 & 17 \\ 0 & 0 & 1 & 1 & 16 \\ 1 & 1 & 0 & 0 & 16 \end{bmatrix} \quad P_{15}: \begin{bmatrix} 0 & 0 & 1 & 1 & 17 \\ & & & & \\ 0 & 1 & 1 & 0 & 17 \\ 1 & 0 & 0 & 1 & 17 \end{bmatrix}$$

Figure 3.7: The generations in form of a matrix

is to compute the answer in stages 5, 8 and 10. These three stages correspond to the following stages of filling the generation matrix (see figure 3.8):

A. Starting with generation P_0 the generations $P_1,\ldots,P_g$ are produced by applying the linear recursive rule bottom-up. This corresponds to stage 5 of section 3.4.1.

B. A d-step rule r^d can be generated from the linear recursive rule r. One application of r^d is equivalent to d applications of r:

$$P_{k+d} = r^d(P_k) = \underbrace{r \circ r \circ \ldots \circ r}_{d}(P_k)$$

A bottom-up iteration of r^d using P_g results in P_{g+d}, P_{g+2d}, P_{g+3d}, $\ldots$ so the column corresponding to P_g is filled. So filling the column of P_g can be done by computing a transitive closure (see stage 8 of section 3.4.1).

C. An application of r, r^2, $\ldots$, r^{d-1} to the generations of the column of P_g respectively fills the rest of the matrix. This corresponds to stage 10 of the algorithm. Alternatively r can be applied $(d-1)$ times to the column generated in stage B.

Parallel Aspects of the Algorithm

There are several possibilities for parallel computations:

1. The most natural and the most effective parallelism occurs in stage B. where an algorithm to compute the transitive closure is applied. On one hand a parallel algorithm can be

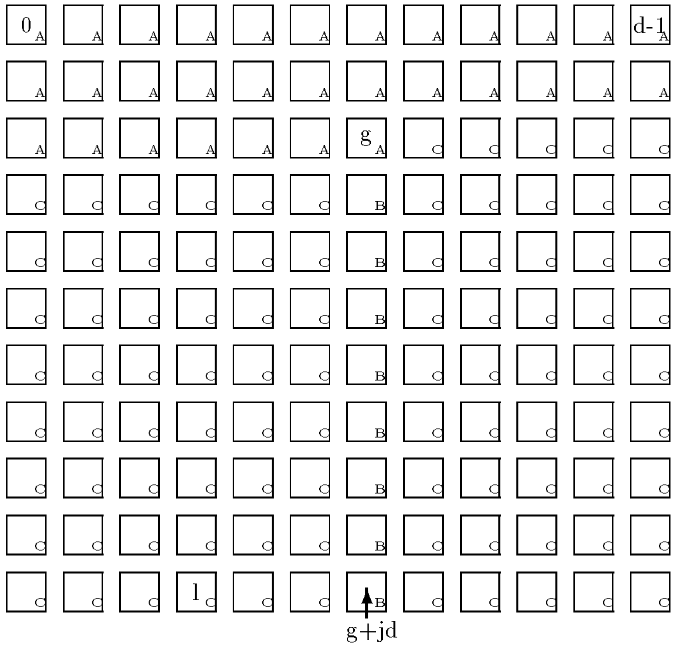


Figure 3.8: Filling of the matrix by the algorithm of Jagadish. The letter indicates the stage by which the generation is produced.

used and on the other hand this algorithm can be executed in parallel for the partitions Q_{x_i} of Q. The critical point is that the number of transitive closures i. e. the number of independent computations can be very large.

2. The applications of the rules r, r^2, ..., r^{d-1} in stage C. to the column of P_g are independent, so $(d-1)$ computations running in parallel are possible.

3. Each (bottom-up) application of the linear recursive rule r or its d-th iterate r^d (which is also a linear recursive rule) to a relation (say Q), i.e. the execution of $r(Q)$ respectively $r^d(Q)$ is itself a (data) parallel computation: $r(Q)$ means that r is applied to each tuple of Q which can be done independently. It is an application of the same code to different data (i.e. data parallelism).

Remarks

- Obviously the parallelism described in 1. and 3. is much more effective than that of 2.
- The number T of transitive closures that can be computed in parallel (see aspect 1.) depends on the number m of instances x_i of X' ($X' \subseteq X$) in the relation Q[5]; it is $T \leq m$. Some partitions Q_{x_i}[6] of Q can be omitted if x_i does not appear in any tuple of the initial

[5] Remember the general basic form (3.2) $p(X,Z)\ :-\ p(X,Y),\ q(X',Y,Z).$ on page 21.
[6] Remember that $Q_{x_i} := \sigma_{X'=x_i}(Q)$

relation E (i. e. $\sigma_{X'=x_i}(E) = \emptyset$) – in this case $T < m$ holds.

If m is very large there will be a large number of partitions Q_{x_i} of Q. This implies that (in general) the Q_{x_i}'s will be really small so that the number of steps to compute the transitive closure of a Q_{x_i} will be small.

- The effort of the parallelism of 2. is restricted to $(d-1)$ parallel computations. In reality d will not be very large in most cases so the grade of parallelism will not be very high normally.

3.4.4 A Variation of the Algorithm

Using the model of the generation matrix other strategies of filling the matrix can be designed, for example the following variation on the algorithm: stage B produces one column of the matrix by computing the transitive closures to get Q^{**} which will be joined with P_g (see stage 8) to produce the final result for the column. In fact the same can be done to get the other $d-2$ columns. All we need is to join Q^{**} with the generations $P_{g+1}, \ldots, P_{g+d-1}$ respectively.

$\bar{\text{A}}$. Compute generations $P_1, P_2, \ldots, P_{g+d-1}$ i. e. the top rows of the generation matrix.

$\bar{\text{B}}$. Use the generations $P_g, P_{g+1}, \ldots, P_{g+d-1}$ to fill the respective column by applying a transitive closure algorithm. This is in fact a parallel execution of stage B. of section 3.4.3.

Chapter 4

Parallel Transitive Closure Algorithms

The algorithm of Jagadish *et al.* which was described in chapter 3 involves apart from relational database operators only an additional transitive closure operation. Therefore the focus of this chapter is to investigate parallel algorithms for transitive closure especially those that can be effectively implemented on a data parallel machine.

4.1 Transitive Closure by Boolean Matrix Multiplication

4.1.1 The Algorithm

The idea of this method (see [Akl 89]) is to use the adjacency matrix of a graph to compute the *transitive closure* of a graph $G = (V, E)$ where

$$V = \{v_1, v_2, \ldots, v_n\}$$

is the set of vertices and

$$E = \{(v_i, v_j) \mid \text{there is an edge from } v_i \text{ to } v_j\ \}$$

is the set of edges of the graph. Using E an $n \times n$ boolean matrix $A = (a_{ij})$ – called the *adjacency matrix* – can be derived:

$$a_{ij} = \begin{cases} 1 & \text{iff} \quad (v_i, v_j) \in E \\ 0 & \text{otherwise} \end{cases}$$

So A contains exactly the paths of length 1. It can be assumed that there is a path of length 0 from each vertex v to itself. So there is an extended set $\bar{E}$ of edges:

$$\bar{E} = E \;\cup\; \{(v,v) \mid v \in V\}$$

Therefore an *extended adjacency matrix* $\bar{A} = (\bar{a}_{ij})$ of G is defined by

$$\bar{a}_{ij} = \begin{cases} 1 & \text{iff} \quad (v_i, v_j) \in \bar{E} \\ 0 & \text{otherwise} \end{cases}$$

Normally a matrix multiplication $C = A * B$ of two $n \times n$ matrices $A = (a_{ij})$ and $B = (b_{ij})$ is defined by

$$c_{ij} = \sum_{k=1}^{n} a_{ik} * b_{kj}$$

According to this definition a boolean matrix multiplication is defined by

$$c_{ij} = \bigvee_{k=1}^{n} a_{ik} \wedge b_{kj} \tag{4.1}$$

so the scalar addition is replaced by a logical OR and the scalar multiplication is replaced by a logical AND operation.

In order to exploit properties of the adjacency matrix (or its extended version) of a graph G in the computation of the transitive closure of G, the two following theorems are needed.

Theorem 2 *If A is the adjacency matrix of a graph $G = (V, E)$ then A^k holds the paths of G that are of length k $(k \geq 0)$.*

Proof by Induction:

Base Case: $k = 0$: So $A^0 = I_n$ (this is the $n \times n$ identity matrix which has the diagonal elements set to 1 and all other elements set to 0). Here the theorem holds because only the diagonal is set to 1 which indicates only paths from a vertex to itself. These paths are of length 0 (by definition).

Assumption: The theorem holds for all $k \leq k_0$ for some $k_0 \geq 0$.

Inductive Step: $k = k_0 + 1 \;\;\Rightarrow\;\; A^k = A^{k_0+1} = A * A^{k_0} \;\;\Rightarrow\;\;$ an element a^k_{ij} of A^k is computed by

$$a^k_{ij} = \bigvee_{l=1}^{n} a_{il} \wedge a^{k_0}_{lj}$$

according to (4.1). Because of the assumption the theorem holds for k_0:

$$a^{k_0}_{lj} = \begin{cases} 1 & \text{iff there is a path of length } k_0 \text{ from } v_l \text{ to } v_j \\ 0 & \text{otherwise} \end{cases}$$

and because of the definition of A:

$$a_{il} = \begin{cases} 1 & \text{iff there is an edge from } v_i \text{ to } v_l \\ 0 & \text{otherwise} \end{cases}$$

Therefore

$$a_{il} \wedge a_{lj}^{k_0} = \begin{cases} 1 & \text{iff there is a path of length } (k_0+1) \text{ from } v_i \text{ to } v_j \text{ via } v_l \\ 0 & \text{otherwise} \end{cases}$$

So if all possible neighbours v_l of v_i are tested we get

$$a_{ij}^k = \bigvee_{l=1}^{n} a_{il} \wedge a_{lj}^{k_0} = \begin{cases} 1 & \text{iff there is a path of length } (k_0+1) \text{ from } v_i \text{ to } v_j \\ 0 & \text{otherwi+se} \end{cases}$$

So the theorem holds for $k = k_0 + 1$. $\square$

Theorem 3 *If $\bar{A}$ is the extended adjacency matrix of a graph $G = (V, E)$ then $\bar{A}^k$ holds the paths of G that are of length k or less ($k \geq 0$).*

Proof by Induction:

Base Case: $k = 0$: So $\bar{A}^0 = I_n$ Here the theorem holds because only the diagonal is set to 1 which indicates only paths from a vertex to itself. These paths are of length 0 (by definition).

Assumption: The theorem holds for all $k \leq k_0$ for some $k_0 \geq 0$.

Inductive Step: $k = k_0+1 \Rightarrow \bar{A}^k = \bar{A}^{k_0+1} = \bar{A} * \bar{A}^{k_0} \Rightarrow$ an element $\bar{a}_{ij}^k$ of $\bar{A}^k$ is computed by

$$\bar{a}_{ij}^k = \bigvee_{l=1}^{n} \bar{a}_{il} \wedge \bar{a}_{lj}^{k_0}$$

according to (4.1). Because of the assumption the theorem holds for k_0:

$$\bar{a}_{lj}^{k_0} = \begin{cases} 1 & \text{iff there is a path of length } k_0 \text{ or less from } v_l \text{ to } v_j \\ 0 & \text{otherwise} \end{cases}$$

and because of the definition of $\bar{A}$:

$$\bar{a}_{il} = \begin{cases} 1 & \text{iff there is an edge from } v_i \text{ to } v_l \text{ or } i = l \\ 0 & \text{otherwise} \end{cases}$$

Therefore

$$\bar{a}_{il} \wedge \bar{a}_{lj}^{k_0} = \begin{cases} 1 & \text{iff there is a path of length } k_0 \text{ or less from } v_l \text{ to } v_j \text{ and } i = l \\ & \text{or there is a path of length } k_0 \text{ or less from } v_l \text{ to } v_j \text{ and an} \\ & \text{edge from } v_i \text{ to } v_l \\ 0 & \text{otherwise} \end{cases}$$

$$= \begin{cases} 1 & \text{iff there is a path of length } k_0 \text{ or less from } v_i \text{ to } v_j \text{ or} \\ & \text{there is a path of length } (k_0+1) \text{ or less from } v_i \text{ to } v_j \text{ via } v_l \\ 0 & \text{otherwise} \end{cases}$$

So if all possible neighbours v_l of v_i are tested we get

$$\bar{a}_{ij}^{k} = \bigvee_{l=1}^{n} \bar{a}_{il} \wedge \bar{a}_{lj}^{k_0} = \begin{cases} 1 & \text{iff there is a path of length } (k_0+1) \text{ or less} \\ & \text{from } v_i \text{ to } v_j \\ 0 & \text{otherwise} \end{cases}$$

So the theorem holds for $k = k_0 + 1$. □

Two algorithms to compute the transitive closure can be derived from the theorems (see figures 4.1 and 4.2). Remember that n is the number of vertices of the corresponding graph so A, B, C are $n \times n$ matrices respectively.

- **Algorithm 1** collects all paths given by $A, A^2, \ldots, A^n$ of length $1, 2, \ldots, n$ respectively to get the connectivity matrix C of the graph. The *connectivity matrix* $C = (c_{ij})$ of a graph G is defined by

$$c_{ij} = \begin{cases} 1 & \text{iff there is a path of length} \geq 1 \text{ from } v_i \text{ to } v_j \\ 0 & \text{otherwise} \end{cases}$$

  and holds the *transitive closure* of G (A respectively).

- **Algorithm 2** computes the extended connectivity matrix $\bar{C} = \bar{A}^n$. The *extended connectivity matrix* $\bar{C} = (\bar{c}_{ij})$ of a graph G is defined by

$$\bar{c}_{ij} = \begin{cases} 1 & \text{iff there is a path of length} \geq 0 \text{ from } v_i \text{ to } v_j \\ 0 & \text{otherwise} \end{cases}$$

  and holds the *reflexive and transitive closure* of G (A respectively).

In the next chapter a parallel (boolean) matrix multiplication algorithm with time complexity $O(n)$ is presented. Although parallel matrix multiplication can be done in $O(\log n)$ it is

```
{A is the adjacency matrix of a graph}
B = A;
C = A;
for k=2 to n do begin
      B = B * A;        {* is a boolean matrix multiplication}
      C = C ∨ B;        {this is an elementwise ∨}
   end;
{C holds the connectivity matrix of the graph}
```

Figure 4.1: Algorithm 1 to compute the transitive closure of a graph.

```
{Ā is the extended adjacency matrix of a graph}
C̄ = Ā;
for k=1 to ⌈log₂ n⌉ do
      C̄ = C̄ * C̄;
{C̄ holds the extended connectivity matrix of the graph}
```

Figure 4.2: Algorithm 2 to compute the reflexive and transitive closure of a graph

more realistic to assume the $O(n)$ algorithm because the $O(\log n)$ algorithm uses n^3 processors. Remember that n is the number of vertices of the graph G or the size of the corresponding adjacency matrix A. The latter two originate in some arbitrary database relation so n is likely to be a large number say e. g. $n = 1000$ which implies that $n^3 = 1\,000\,000\,000$ processors would be necessary. Even if the concept of virtual processors is used by the operating system of the parallel machine on which the algorithm is implemented the performance of the code will suffer from the overload of a single physical processor by a large number of virtual processors. In the case of the Connection Machine figures show that even the $O(n)$ matrix multiplication suffers from too many virtual processors per physical processor if n is large (see tables 4.1, 4.2 and 4.3 for example).

So if matrix multiplication is assumed to be of time complexity of $O(n)$ then algorithm 1 is of $O(n^2)$ and algorithm 2 of $O(n \log n)$.

As mentioned before n may be very large so algorithm 1 is unusable: e. g. for $n = 1024$ and an implementation of the matrix multiplication algorithm of section 4.1.2 on a Connection Machine (using 16384 physical processors a boolean matrix multiplication of two 1024×1024 matrices takes 3.48 seconds – see table 4.3) it takes 3.48 seconds * 1024 (number of runs of the loop's body) $\approx$ 3564 seconds $\approx$ 1 hour.

The problem of algorithm 2 is that the extended connectivity matrix $\bar{C} = (\bar{c}_{ij})$ is computed:

$$\bar{c}_{ij} = \begin{cases} 1 & \text{iff there is a path of length} \geq 0 \text{ from } v_i \text{ to } v_j \\ 0 & \text{otherwise} \end{cases}$$

$$= \begin{cases} 1 & \text{iff there is a path of length} \geq 1 \text{ from } v_i \text{ to } v_j \text{ or } i = j \\ 0 & \text{otherwise} \end{cases}$$

The problem is that in the case of the diagonal elements (which are all set to 1 in the extended connectivity matrix) it is not known whether the 1 is set because there is a path of length ≥ 1 or because it was set right at the beginning. For the purpose of computing the transitive closure (and not the *reflexive* and transitive closure) of a database relation only those 1's in the diagonal are of any interest that are set because there is a path of length ≥ 1. If the corresponding graph G of the database relation is known to be acyclic then algorithm 2 can be used because the connectivity matrix C cannot hold any 1 in the diagonal because there is no path of a length ≥ 1 from any vertex to itself so[1]

$$C = \bar{C} \text{ xor } I_n$$

4.1.2 A Boolean Matrix Multiplication Algorithm

The algorithm that is presented in this section is based on the matrix multiplication algorithm introduced by [Can 69]. Johnnson showed its suitability for parallel machines based on a hypercube architecture ([Joh 87], [Joh 89]).

The idea of this matrix multiplication algorithm[2] is to develop the resulting matrix $C = A * B$ in an $n \times n$ mesh of processors each developing one element c_{ij} of C. Note that the architecture of the processors corresponds to that of the matrices. Therefore the coordinates of a processor will be referred to by i and j in the same way as to the coordinates of some matrix element say a_{ij}.

By definition

$$c_{ij} = \bigvee_{k=1}^{n} a_{ik} \wedge b_{kj}$$

so the idea is to hold in each step in each processor p_{ij} two elements a_{ik} and b_{kj} for some k that have to be combined once by an AND operation to compute c_{ij} and to send them afterwards to a neighbour processor of p_{ij} that will need a_{ik} and b_{kj} next. The trick is to provide each processor in each step with a pair (a_{ik}, b_{kj}) so that the processors can work independently in parallel. This can be done without duplicating any value of A or B which would cause further communication among the processors and therefore would increase the costs.

Assume that the n^2 (virtual) processors are organised in an $n \times n$ matrix. Each processor p_{ij} holds the elements a_{ij} and b_{ij}.

- If the rows of A are numbered by $1, 2, \ldots, n$ then shift the elements of the i-th row $(i-1)$

[1] The XOR operator for two matrices is defined to perform an elementwise XOR operation.

[2] For the purpose of computing a transitive closure it is only necessary to deal with symmetric matrices i. e. $n \times n$ matrices. So we will only refer to this special case neglecting the general case of matrix multiplication.

positions to the left (wrapping elements that would be pushed out of the processor matrix around to fill the last $(i-1)$ positions of the resulting row).

- If the columns of B are numbered by $1, 2, \ldots, n$ then shift the elements of the j-th column $(j-1)$ positions to the top (wrapping elements that would be pushed out of the processor matrix around to fill the $(j-1)$ bottom positions of the resulting column).

See figures 4.3 and 4.4 for the result of this 'shifting'. After that the two matrices A, B are ready for computation. If the 'prepared' matrices A and B are overlapped like in figure 4.5 it can be seen that each processor p_{ij} holds exactly two elements a_{ik} and b_{kj} for some k.
After combining the two elements a_{ik} and b_{kj} by an AND operation (in each processor) and performing an OR with a variable c_{ij} that was initialised to 0 (*false*) at the beginning a_{ik} and b_{kj} can be sent to neighbour processors in the following way:

- All a_{ik} are sent to the left neighbour of the processor that held a_{ik} respectively. As in the preparation of A border processors in the first column of the processor matrix send to the processors of the last column of the processor matrix.

- All b_{kj} are sent to the top neighbour of the processor that held b_{kj} respectively. As in the preparation of B border processors in the first row of the processor matrix send to the processors of the last row of the processor matrix.

Such a step has to be done n times so that each processor p_{ij} gets all pairs (a_{ik}, b_{kj}) for $k = 1, 2, \ldots, n$ and produced the corresponding AND and OR so that each processor p_{ij} holds finally the value c_{ij} in one of its registers.

The algorithm is summarised in figure 4.6: n^2 processors are used, organised in an $n \times n$ matrix. Each processor has 3 registers r^a_{ij}, r^b_{ij} and r^c_{ij} which are initialised by a_{ij}, b_{ij} and 0 (or *false*) respectively.

The function t_n is used to compute the appropriate address if the argument of t_n is not positive i. e. an element would be sent to a place beyond the border. In this case an address is computed to wrap the element around. t_n can be defined like this

$$t_n(x) = \begin{cases} x & \text{if } 1 \leq x \leq n \\ x+n & \text{if } -n < x \leq 0 \\ \perp & \text{otherwise} \end{cases}$$

4.1.3 Performance Results for Boolean Matrix Multiplications on a CM-200

Three different boolean matrix multiplication algorithms have been implemented. The advantage of implementing a matrix multiplication instead of using the implementation of the Connection

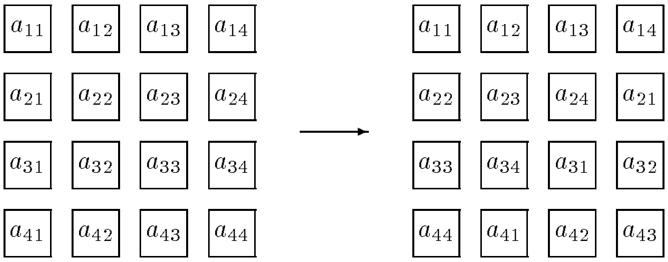


Figure 4.3: Preparing a 4×4 matrix A for computation.

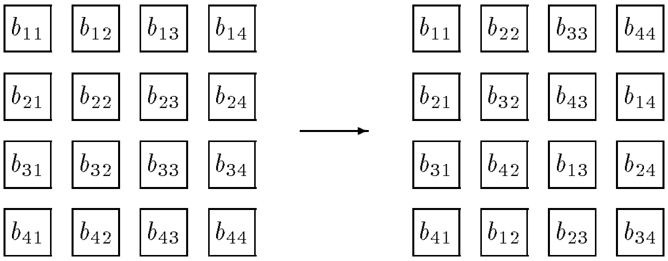


Figure 4.4: Preparing a 4×4 matrix B for computation.

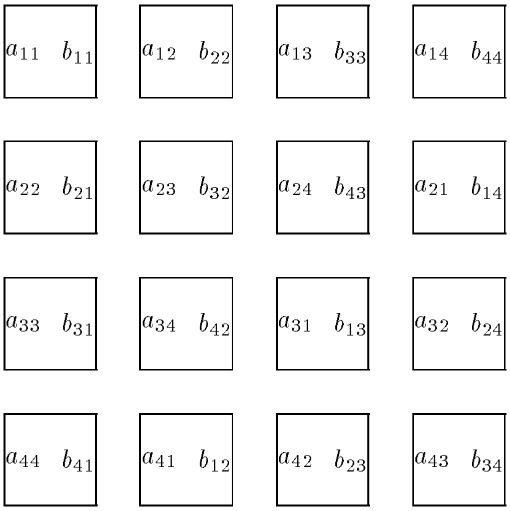


Figure 4.5: Overlapping the 'prepared' matrices A and B.

$\{\ \forall i \forall j\ :\ r^a_{ij} = a_{ij}\ \wedge\ r^b_{ij} = b_{ij}\ \wedge\ r^c_{ij} = 0\ \}$

{ prepare matrices A and B }
for i=1 **to** n **do in parallel**
    for j=1 **to** n **do in parallel**
        begin
            send value of r^a_{ij} to $r^a_{i\,t_n(j-i+1)}$
            send value of r^b_{ij} to $r^b_{t_n(i-j+1)\,j}$
        end

{ perform matrix multiplication }
for k=1 **to** n **do**
    for i=1 **to** n **do in parallel**
        for j=1 **to** n **do in parallel**
            begin
                $r^c_{ij} = r^c_{ij}\ \vee\ (r^a_{ik}\ \wedge\ r^b_{kj})$
                send value of r^a_{ij} to $r^a_{i\,t_n(j-1)}$
                send value of r^b_{ij} to $r^b_{t_n(i-1)\,j}$
            end
$\{\ \forall i \forall j\ :\ r^c_{ij} = \bigvee_{k=1}^{n} (a_{ik}\ \wedge\ b_{kj})\ \}$

Figure 4.6: Boolean Matrix Multiplication Algorithm

Machine Scientific Software Library (CMSSL) is that the CMSSL-implementation does not exploit the fact that the matrix elements are only boolean values: the matrix multiplication is done on the level of floating point arithmetic instead of boolean arithmetic. The CMSSL floating point matrix multiplication of two 128×128 matrices (using 16384 physical processors) takes about 4 seconds elapsed time whereas a boolean matrix multiplication for the same matrices (also using 16384 processors) takes about 0.04 seconds elapsed time.

Additionally it is a waste of memory to hold a boolean matrix in the same way as a matrix of real numbers. An advantage of the CM is that memory can be allocated bitwise so the boolean matrices can be stored in a way they should be stored – without any waste of memory.

All algorithms that have been implemented use an $n \times n$ mesh i. e. n^2 (virtual) processors. Although all algorithms are of time complexity $O(n)$ there are large differences in the performance results of their implementations "mmult 1", "mmult 2a", "mmult 2b" and "mmult 3". We will show the reasons for this behaviour:

Algorithm 1 [3] (see "Mesh Matrix Multiplication" in [Akl 89]) computes the resulting matrix $C = (c_{ij})$ in $3n - 2$ steps by developing the elements c_{ij} in the corresponding processor p_{ij} of the mesh. In the first step only processor p_{11} is active, in the second step processors p_{11}, p_{12}, p_{21} are active, in the third step p_{11}, p_{12}, p_{21}, p_{13}, p_{22}, p_{31} are active and so on. For details see [Akl 89].

[3] Its implementation is called "mmult 1"

The essential part of "mmult 1" consists of a loop that runs $3n-2$ times. The body is executed in constant time but involves not the most but the most complicated communication among processors. This fact can be seen by table 4.1: The greater the overload of a physical processor by a large number of virtual processors the smaller the gap between "mmult 1" and "mmult 2a" ("mmult 2b" respectively) becomes although "mmult 2a" and "mmult 2b" involve the most communication.

Additionally algorithm 1 suffers from the fact that its loop runs $3n-2$ times whereas the loops of algorithms 2 and 3 run only n times. This explains the figures of table 4.2 and 4.3.

Algorithm 2 [4] The idea of this algorithm is to use the transposed matrix B^T of B and push it over A to compute in each step n elements of the resulting matrix C. Computation involves an elementwise AND and an OR scan along the rows of the processor matrix. After this scan the results of the n computed elements are found in the n processors of the first column of the processor matrix and have to be sent to the appropriate place afterwards.

Although we avoided to tell any details this description shows the high communication costs of this algorithm. Table 4.1 shows that the loop-bodies of "mmult 2a" and "mmult 2b" take the most time.

On the other hand the body of the loop is performed only n times which makes "mmult 2a" and "mmult 2b" faster than "mmult 1" (see tables 4.2 and 4.3).

Algorithm 3 [5] (see [Can 69] or [Joh 87]) was discussed in section 4.1.2. Its most important advantage are the low communication costs which can be seen by table 4.1.

4.2 A Parallel Warshall-derived Algorithm for Transitive Closure

4.2.1 The Algorithm

Starting from the point of view that every successor of a vertex v_j of a graph $G = (V, E)$ is also a successor of a vertex v_i if there is an edge from v_i to v_j (i. e. $(v_i, v_j) \in E$) and that each row i of the adjacency matrix $A = (a_{ij})$ of G represents the list of direct successors of the corresponding vertex v_i some strategies can be derived in which order successor lists may be added to other successor lists if the corresponding vertices are successors. More formally this action can be described as: if $a_{ij} = 1$ then add the successor list of v_j to that of v_i[6] This action

[4] There are two implementations of algorithm 2 which are called "mmult 2a" and "mmult 2b". The difference is that 2a uses *get* operations for communicating among processors whereas 2b uses only *send* operations. Although *send* operations are cheaper in general 2a performs better for some values of n (see tables 4.1 and 4.2).

[5] Its implementation is called "mmult 3"

[6] If a boolean matrix is used as the data structure to represent successor lists adding a successor list to another is in fact applying an OR operation to the corresponding elements of the rows that represent the successor lists.

Loop Body of "mmult 1" = 100, VP = virtual processors per physical processor.

n	VP	mmult 1	mmult 2a	mmult 2b	mmult 3
128	1	100	211	164	26
128	2	100	216	172	22
256	4	100	169	172	17
256	8	100	158	161	15
512	16	100	130	149	13
512	32	100	125	145	12
1024	64	100	111	139	11
1024	128	100	108	135	11
2048	256	100	108	133	9
2048	512	100	115	128	9

Table 4.1: Performance Results of the Loop Bodies

Performance of "mmult 1" = 100, VP = virtual processors per physical processor.

n	VP	mmult 1	mmult 2a	mmult 2b	mmult 3
128	1	100	70	55	10
128	2	100	73	58	9
256	4	100	56	58	7
256	8	100	53	55	6
512	16	100	43	50	5
512	32	100	42	49	5
1024	64	100	37	46	4
1024	128	100	36	45	4
2048	256	100	36	44	3
2048	512	100	39	44	3

Table 4.2: Performance Results of the Matrix Multiplication Implementations

16384 processors on a Connection Machine CM-200 used, elapsed time in seconds.

n	VP	mmult1	mmult2a	mmult2b	mmult3
128	1	0.35	0.26	0.19	0.04
256	4	1.78	1.02	1.00	0.12
512	16	11.82	5.50	5.90	0.59
1024	64	88.20	34.48	41.69	3.48
2048	256	692.29	262.52	317.08	23.66

Table 4.3: Performance Results of the Boolean Matrix Multiplication Implementations

```
{ C holds the adjacency matrix of a graph G }
for j = 1 to n do
    for i = 1 to n do in parallel
        process c_ij
{ C holds the connectivity matrix of the graph G }
```

Figure 4.7: A Parallel Warshall-derived Transitive Closure Algorithm

is called *processing* the element (i, j) (a_{ij} respectively).

Warshall showed that there are several strategies to transform the adjacency matrix A to the connectivity matrix C. He also showed that every strategy is correct that satisfies the two following constraints:

Constraint 1: for all i, j, k processing of element (i, k) precedes processing of element (i, j) iff $k < j$

Constraint 2: for all i, j, k processing of element (j, k) precedes processing of element (i, j) iff $k < j$

A less formal description of the constraints involving the fact that a boolean matrix is used to represent the successor lists is

1. Elements in the same row are processed from left to right.

2. Element (i, j) can be processed if row j is processed up to $(j - 1)$-th element.

See [Agr 90] for that proof that every strategy of *processing* the elements that satisfies constraint 1 and 2 is correct. See [Agr 90] also for a more detailed discussion of Warshall-derived transitive closure algorithms.

Figure 4.7 shows a parallel Warshall-derived transitive closure algorithm. Starting with a matrix C initialised to the adjacency matrix A of a graph G it processes the elements of C columnwise from left to right. Elements of a column are processed in parallel so the algorithm takes n (parallel) steps to process the n columns of C.

The organisation of the n^2 processors that are used is the same as in the previous chapter so an $n \times n$ matrix of processors is used each having a 1-bit register to hold the value of c_{ij}.

Figure 4.8 shows an implementation of the algorithm in C*. The matrix B of this implementation provides copies of the j-th row of C where row j is needed. Only the creation of B involves communication among the processors. The costs of communication depend on the number of copies of row j that is necessary i. e. communication depends on the actual data. Table 4.4 shows this data dependency although the effect of having a less filled matrix is not saving as many costs as expected.

To prove the correctness of the algorithm we will show the following

```
shape [MAX_N][MAX_N]matrixshape;
typedef bool:matrixshape boolmatrix;

boolmatrix TRANSITIVE_CLOSURE ( boolmatrix C, unsigned n )
  {
    /* matrixshape must be active */
    unsigned j;
    boolmatrix B;

    for ( j = 0; j < n; j = j + 1 )
        where ( [pcoord(0)][j]C )
          { B = [j][pcoord(1)]C;    /* B provides copies of the j - th row */
            C = C || B;
          }
    return C;
  }
```

Figure 4.8: C* Program for the Parallel Warshall-derived Transitive Closure Algorithm

S/V = (direct) successors per vertex;
F = percentage of (adjacency) matrix elements that are set to 1
16384 processors on a CM-200 used; CM CPU time in seconds

S/V	0	160	320	480	640	800	960
F	0	16	31	47	63	78	94
Time	108.38	145.53	146.56	147.09	147.62	148.16	148.36

Table 4.4: Dependency of the Warshall-derived Algorithm on the filling degree of an underlying 1024×1024 matrix

Lemma 3 *$I \equiv \forall\, 0 \le k < j\ \forall\, 0 \le i < n\ :\ processed(c_{ik})$ is the loop invariant of the algorithm of figure 4.7.*

Proof: Instead of the **for**-loop construction, we will use a **while**-loop for the algorithm of figure 4.7 which leads to the equivalent description

$j = 1$
while $j \le n$ **do begin**
  P_1 : **for** $i = 1$ **to** n **do in parallel**
      process c_{ij}
  P_2 : $j = j + 1$
end

which makes the formal verification more obvious.

I is the invariant of the **while**-loop (and therefore of the **for**-loop of figure 4.7)

- iff $\{I\}\ P_1;\ P_2\ \{I\}$ holds, i. e.
- iff $\{I\}\ P_1\ \{I'\}\ P_2\ \{I\}$ holds.

but this can be shown by[7]

$$I \equiv \forall\, 0 \le k < j\ \forall\, 0 \le i < n\ :\ \text{processed}(c_{ik})$$

P_1 : **for** $i = 1$ **to** n **do in parallel**
    process c_{ij}

$$\forall\, 0 \le k < j\ \forall\, 0 \le i < n\ :\ \text{processed}(c_{ik})\ \wedge\ \forall\, 0 \le i < n\ :\ \text{processed}(c_{ij})\ \equiv\ I'$$
$$\Rightarrow\ \forall\, 0 \le k \le j\ \forall\, 0 \le i \le n\ :\ \text{processed}(c_{ik})$$
$$\Rightarrow\ \forall\, 0 \le k < (j+1)\ \forall\, 0 \le i < n\ :\ \text{processed}(c_{ik})$$

P_2 : $j = j + 1$

$$\forall\, 0 \le k < j\ \forall\, 0 \le i < n\ :\ \text{processed}(c_{ik})\ \equiv\ I \qquad \square$$

Lemma 4 *Constraint 1 is satisfied by the algorithm.*

[7] Please note that the **for** ... **to** ... **do in parallel** construction is no loop and only sets the context in which the following statement is executed in parallel. In this case this context is that i ranges over $\{1, \ldots, n\}$. In C* the context is set by **where** clauses.

Proof: Let $j = s$ for some s in the body of the **for**-loop of the algorithm of figure 4.7. Constraint 1 for this specific s looks like this

$$\forall\, 0 \le k < s \,\forall\, 0 \le i < n \; : \; \text{processing of } (i, k) \text{ precedes processing of } (i, s)$$

i. e. before c_{ic} is processed all c_{ik} for $0 \le k < s$ and $0 \le i < n$ have to be processed. So

$$\forall\, 0 \le k < s \,\forall\, 0 \le i < n \; : \; \text{processed}(c_{ik}) \tag{4.2}$$

must hold before *process* c_{is} is executed. But this is apparently true because the loop-invariant I (see lemma 3) holds before *process* c_{is} is reached:

$$\forall\, 0 \le l < j \,\forall\, 0 \le i < n \; : \; \text{processed}(c_{il})$$

so in the case that $j = s$

$$\forall\, 0 \le l < s \,\forall\, 0 \le i < n \; : \; \text{processed}(c_{il}) \tag{4.3}$$

holds which is equivalent to (4.2) so (4.2) holds before *process* c_{is} is executed. Therefore constraint 1 is satisfied. □

Lemma 5 *Constraint 2 is satisfied by the algorithm.*

Proof: Let $j = s$ for some s in the body of the **for**-loop of the algorithm of figure 4.7. Constraint 2 for this specific s looks like this

$$\forall\, 0 \le k < c \,\forall\, 0 \le i < n \; : \; \text{processing of } (s, k) \text{ precedes processing of } (i, s)$$

i. e.

$$\forall\, 0 \le k < s \; : \; \text{processed}(c_{sk}) \tag{4.4}$$

must hold before *process* c_{is} is executed. But this is implied by the loop-invariant I which holds before *process* c_{is} is reached (see lemma 3), so

$$\forall\, 0 \le l < j \,\forall\, 0 \le i < n \; : \; \text{processed}(c_{il})$$

holds and for $i = s$

$$\forall\, 0 \le l < s \; : \; \text{processed}(c_{sl}) \tag{4.5}$$

holds which is equivalent to (4.4). Therefore constraint 2 is satisfied. □

Theorem 4 *The algorithm of figure 4.7 is correct.*

Proof: see lemmas 4 and 5. □

```
{ C is initialised by the adjacency matrix of the graph. }
{ The n × n matrix C is split into s^2 m × m submatrices C'_pq }
{ so n is assumed to be a multiple of m so that n = s * m }
{ An element of a submatrix C'_pq is referred to as c^pq_ij }

for q = 1 to s do
   for p = 1 to s do
      for j = 1 to m do
         for i = 1 to m do in parallel
            process c^pq_ij
```

Figure 4.9: Adapted Version of the Parallel Warshall-derived Transitive Closure Algorithms

4.2.2 Improving the Performance for Large Graphs

As seen by table 4.3 overloading a physical processor by a large number of virtual processors leads to a decreased performance. Therefore it is desirable to split computation of a large connectivity matrix (i. e. computation of a transitive closure of a graph with a large number of vertices). This can be done by splitting an $n \times n$ matrix X into smaller $m \times m$ $(m < n)$ matrices X'_{pq}, e.g. a 512×512 matrix can be split into sixteen 128×128 submatrices (or four 256×256 submatrices).

The idea is to reduce parallelism of the algorithm presented in section 4.2.1 to reduce the overload of the physical processors. This can be done by a small change in the algorithm: instead of processing the elements of a whole column of a large matrix C in parallel the column is split into smaller pieces (according to the splitting of C). The pieces are processed in a sequential order but the processing itself is done in parallel. Therefore each parallel processing stage involves only m^2 instead of n^2 (virtual) processors. This adapted version of the algorithm of figure 4.7 is shown in figure 4.9.

Although it has a time complexity of $O(s^2 \cdot m)$ the performance results show that it is faster than the algorithm presented in section 4.2.1. See section 4.3 for performance results.

On the other hand processing an element c^{pq}_{ij} gets more complicated: not only submatrix C'_{pq} but also $C'_{q1}, C'_{q2}, \ldots, C'_{qm}$ have to be accessed.

The most important question is to find an adequate value for m. See table 4.5 for details about this choice for a 1024×1024 matrix: using 8192 processors on a CM-200 $m = 128$ is the optimal solution[8] whereas for 16384 processors it is better to use $m = 256$.

[8] Matrixsizes correspond to sizes of "shapes" on the CM. Therefore the matrixsizes have to obey the same rules as shapesizes which have to be a power of 2.

Performance of the simple Warshall-derived Algorithm = 100
other performances relative to this one

m	simple	128	256	512	1024
8192 processors	100	45	50	76	131
16384 processors	100	57	54	77	132

Table 4.5: Performance Results for different values of m for a 1024×1024 matrix

4.3 Performance Results of the Transitive Closure Algorithms on a CM-200

Tables 4.6 and 4.7 show overall performance results of the transitive closure algorithms discussed in this section. Apparently the algorithm that uses boolean matrix multiplication performs best. Nevertheless remember that it can be only applied if the underlying graph is known to be acyclic. Therefore in the case of a cyclic graph the appropriate version of the Warshall-derived algorithm must be used. Which version to choose depends on the size of the graph and on the number of processors that are available:

- If the graph has 128 or less vertices use the simple Warshall-derived algorithm.
- If the graph has more than 128 vertices use the adapted version of the Warshall-derived algorithm. Depending on the number of (physical) processors that are available (see table 4.5) and the number of vertices choose the value of m.

Tables 4.6 and 4.7 show results of a specific hardware configuration and comparing the two tables it is apparent that the choice of the appropriate transitive closure algorithm depends also on this hardware configuration so no general algorithm for the choice can be given but a skeleton:

- For acyclic graphs use the algorithm based on boolean matrix multiplication.
- For cyclic graphs use a version of the Warshall-derived algorithm:
  - For graphs with a number of vertices less than a certain bound use the simple version.
  - For large graphs use the adapted version choosing an appropriate value of m (which depends on the hardware configuration and the number of vertices)

TC-BMM = Algorithm using boolean matrix multiplication
simple WTC = simple Warshall-derived Algorithm
WTC 128 = adapted Warshall-derived Algorithm with $m = 128$
WTC 256 = adapted Warshall-derived Algorithm with $m = 256$
Performance of "simple WTC" = 100
VP = virtual processors per physical processor

n	VP	TC-BMM	simple WTC	WTC 128	WTC 256
128	2	72	100	132	—
256	8	46	100	85	130
512	32	35	100	57	78
1024	128	30	100	43	51
2048	512	28	100	34	35

Table 4.6: Performance Results of the Transitive Closure Algorithms on a CM-200 using 8192 processors

TC-BMM = Algorithm using boolean matrix multiplication
simple WTC = simple Warshall-derived Algorithm
WTC 128 = adapted Warshall-derived Algorithm with $m = 128$
WTC 256 = adapted Warshall-derived Algorithm with $m = 256$
Performance of "simple WTC" = 100
VP = virtual processors per physical processor

n	VP	TC-BMM	simple WTC	WTC 128	WTC 256
128	1	96	100	135	—
256	4	58	100	103	134
512	16	40	100	74	82
1024	64	32	100	57	54
2048	256	29	100	48	39

Table 4.7: Performance Results of the Transitive Closure Algorithms on a CM-200 using 16384 processors

Chapter 5

Implementation and Performance Evaluation

The algorithm presented in chapter 3, the most suitable transitive closure algorithms of chapter 4 and a parallel relational database system were implemented on a Connection Machine CM-200 with 16384 processors. For an introduction to the hardware architecture and the software environment see section 1.2.

5.1 The Implementation

The algorithm presented in chapter 3, the transitive closure algorithms of chapter 4 and a parallel relational database system were implemented on a Connection Machine CM-200 with 16384 processors. For an introduction to the hardware architecture and the software environment see section 1.2. The architecture of the software is given by figure 5.1: only the relational database system module and the transitive closure processor involve parallel code. These two modules can be seen as a unit because the transitive closure processor supplements the relational database system with an additional operation. The tasks of the modules are as follows:

Query Analyser: This involves a parser for Datalog rules and the analysing technique to compute the substitution graph, the parameters d and g, the d-step rule etc. for further query processing.

Query Processor: This module essentially executes the stages A., B. and C. (see section 3.4.3) to evaluate the query by calling the appropriate operations to access the database. This can be

- evaluating a nonrecursive Datalog rule using the Datalog – relational algebra interface

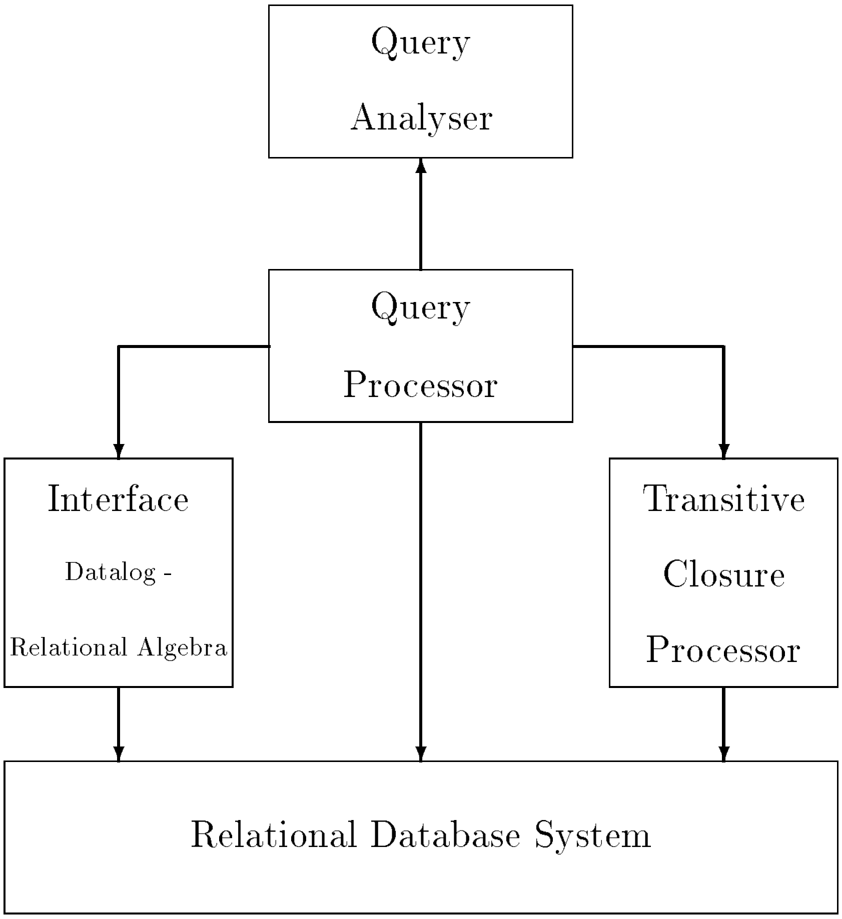


Figure 5.1: Software architecture and the 'uses' relation

- computing the transitive closures of a relation by calling the transitive closure processor giving the partition, source and destination attributes[1].
- calling operations to access the relational database, e. g. loading or storing a relation.

Datalog – Relational Algebra Interface: This module derives Datalog equations from Datalog rules. An evaluation procedure for Datalog equations is provided too.

Transitive Closure Processor: The essential procedure of this module computes the transitive closures of a relation by giving its partition, source and destination attributes. As mentioned above this module can be seen as an extension of a usual relational database system.

Relational Database System: This is a simple parallel implementation of a relational database system.

[1] This refers to the general basic form (3.2) where the relation Q is given by $Q(X', Y, Z)$. X' are the partition, Y the source and Z the destination attributes.

5.2 Performance Evaluation

The sample relations that were used for the evaluation are acyclic, i. e. their corresponding graph representations are acyclic. This is no serious restriction because this implies only that the transitive closure algorithm based on boolean matrix multiplication was used instead of the Warshall-derived algorithm (see chapter 4). Both are of the same time complexity so the overall behaviour of the implementation is not influenced by this choice. However the bottom-up evaluation strategy that was used to compare the results profits from acyclic graphs because every generation P_i holds exactly those tuples that are given by a path of length $i-1$ in the corresponding graph representation (see example 4 in section 2.3.3). Therefore generation P_l ($l =$ length of longest path in the relation Q of the general basic form (3.2)), which is produced by the l-th application of the linear recursive rule (see section 3.1), will be empty and the bottom-up evaluation stops. In the case of cyclic graphs it is not so easy to identify the point when a bottom-up evaluation may stop because there will be no i such that P_i is empty although every tuple of P_i may be in some P_j for $j < i$ which means that further iterations will not produce any new tuple. Therefore the results for the bottom-up evaluation have to be treated with caution as they are obtained within 'ideal' conditions.

5.2.1 Varying the Number of Partitions

As we have seen in example 5 in section 2.3 and also by the introduction of the general basic form (3.2) there is a class of queries that can only be processed by computing a *data-dependent* number of transitive closures. This data-dependency was investigated by processing one query on different starting relations. The query that was used is a simple example of this class, as the linear recursive rule is already in the general basic form:

$$\begin{aligned} p(X_1, X_2, Z) \quad &:- \quad q(X_1, X_2, Z). \\ p(X_1, X_2, Z) \quad &:- \quad p(X_1, X_2, Y),\ q(X_1, Y, Z). \end{aligned}$$

It is easy to see that $d = 1$ and $g = 0$ so the linear recursive rule is also the d-step rule.

The sample relations for Q are all of size $S = 2000$ tuples. We let SP denote the number of partitions of Q. The structure of a sample relation Q can be seen in figure 5.2. Each partition Q_i corresponds to a simple acyclic graph with the longest path $LP = (S \textit{ div } SP)$. In the case that $(SP \cdot LP) < S$ some tuples are added to each partition Q_i. These additional tuples (at most one per partition) do not affect the graphs as they do not change the length LP of the longest paths.

SP was varied over three different ranges as can be seen in figures 5.3, 5.4 and 5.5. The effects of increasing the value of SP affects the behaviour of the algorithm in the following way:

1. As the number of Q_i increases the number of transitive closures that have to be computed grows (remember: one per Q_i). The larger is the number of partitions Q_i, the more the effort that has to be spent on computing these partitions. This can be seen by figure 5.6: for $SP \geq 250$ more than 80% of the elapsed time to process the query was spent on this step. The strong incline of the total processing time for large values for SP is essentially caused by this fact.

2. As the size of Q stays constant the average size of the partitions Q_i decreases. Therefore the graphs corresponding to the Q_i becomes smaller, i. e. their sets of vertices and edges becomes smaller. Their respective transitive closure are also smaller. For example the time to transform the table representation of a relation to a boolean matrix representation (which is necessary to apply one of the transitive closure algorithms in chapter 4) and vice versa decreases for increasing SP values although the number of transformations increases (see figure 5.7).

3. Because the graphs become smaller their respective longest path becomes shorter. This implies a shorter time to compute the transitive closures which results in a decreasing total elapsed time. This effect and that of 2. can be seen in figures 5.3 and 5.4. For large values of SP this computational advantage is overcompensated by the effect described in 1. (see figure 5.4 for $SP > 50$ or figure 5.5).

4. In contrast a bottom-up evaluation can profit if the maximum of the longest paths decreases which is *not* necessarily implied by an increasing SP. However the sample relations that were used had this property so a bottom-up evaluation of the query is much faster as the number of joins involved is proportional to the maximum longest path. As this maximum was given by $LP = (S \ div \ SP)$ it is obvious that the number of joins decreases for an increasing SP. This can be seen in figure 5.4.

5.2.2 Varying the Diameter d

Regarding the processing algorithm described in section 3.4.3, it is obvious that the time that has to be spent on stage C. grows with an increasing value for d: applying the linear recursive rule r $(d-1)$ times means computing several joins. The number of joins[2] is in fact proportional to d. Therefore the influence of this fact on the total processing time for different queries was investigated.

In contrast to SP the diameter d is a *data-independent* parameter which can be obtained by analysing the query without involving any information about the actual data. Therefore a

[2] Remember that the Datalog equation for r involves $k-1$ joins if k is the number of literals in the body of r that have predicate symbols of ordinary predicates. See section 2.2 for details.

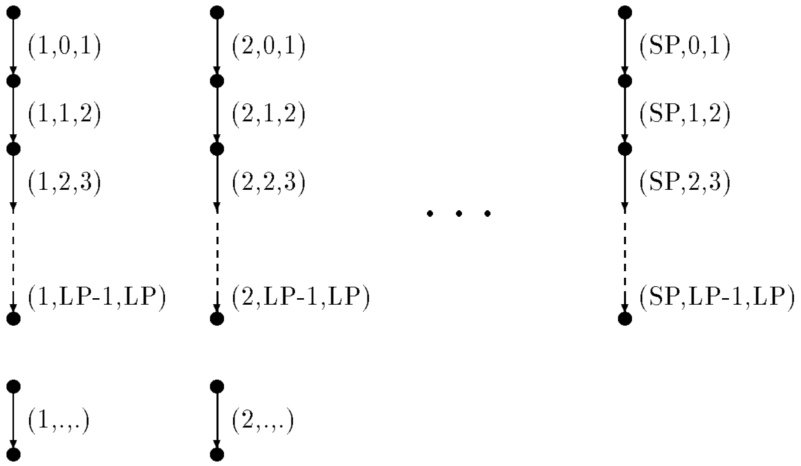
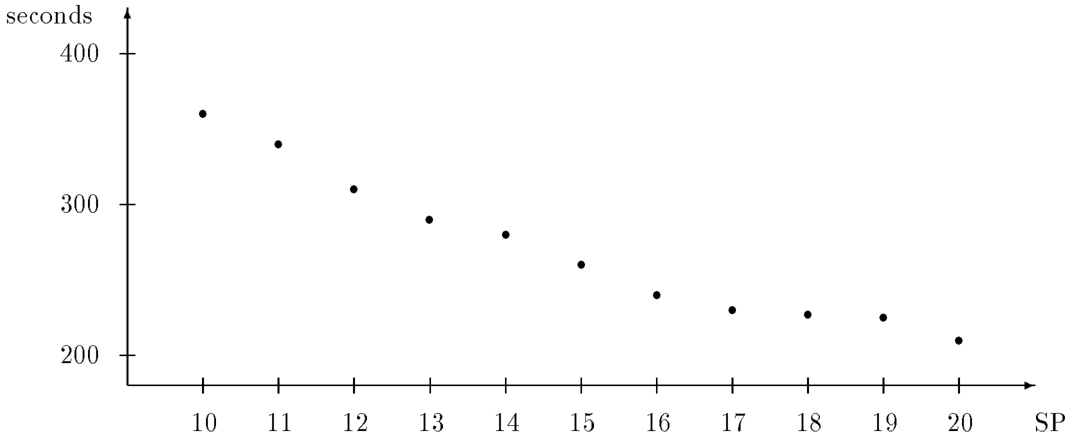


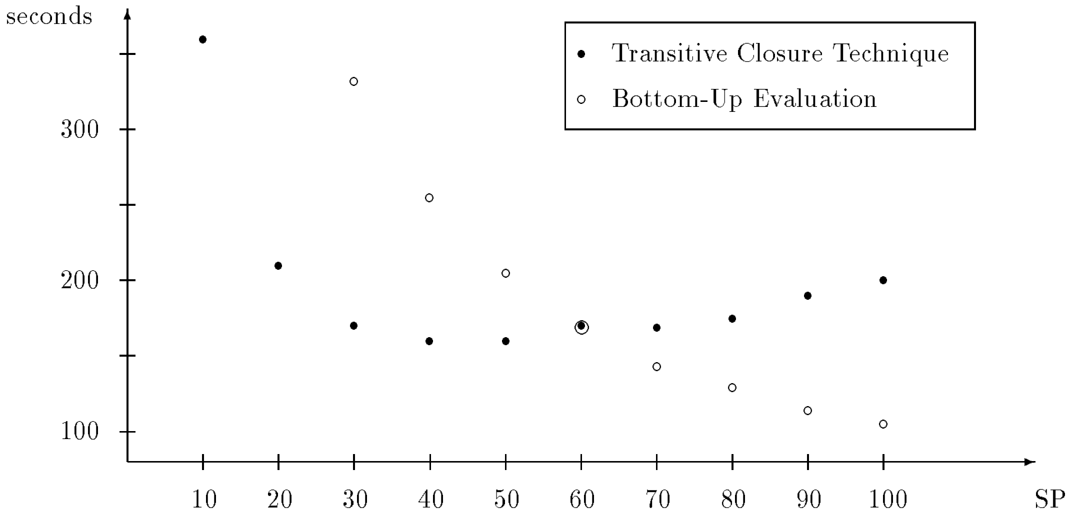
Figure 5.2: The structure of the sample relations for varying over SP.



Elapsed Processing Times in Seconds											
SP	10	11	12	13	14	15	16	17	18	19	20
Total	357.7	344.2	309.5	289.6	276.5	261.0	239.2	231.0	221.7	215.2	209.4
TC	205.1	202.9	180.0	170.4	164.3	157.9	139.5	133.5	131.0	127.7	123.4
Partitions	12.2	13.6	15.2	15.8	17.1	18.4	19.5	20.9	22.2	23.3	24.5
Transforming	179.9	175.6	149.5	138.3	130.2	120.9	114.1	106.6	102.4	97.7	91.9

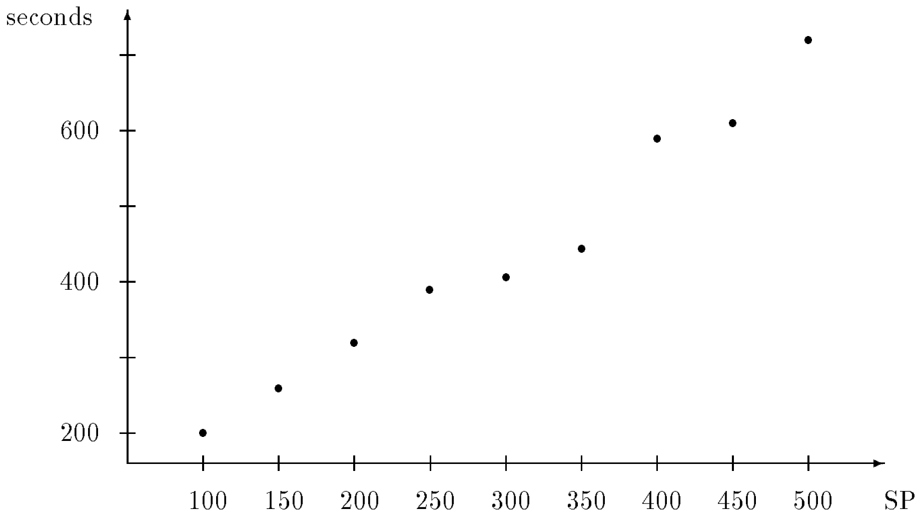
Total = Total Processing Time
TC = Total time spent on computing transitive closures (includes partition and transforming times)
Partitions = Time spent on computing the partitions Q_i
Transforming = Time spent on transforming the table to the boolean matrix representation and vice versa

Figure 5.3: Time Performance (1) for varying over SP



Elapsed Processing Times in Seconds										
SP	10	20	30	40	50	60	70	80	90	100
Total	357.7	209.4	170.5	161.6	158.6	169.4	169.6	176.4	192.1	199.3
TC	205.1	123.4	108.0	108.7	114.8	127.4	134.5	142.9	157.7	168.1
Partitions	12.2	24.5	36.6	49.4	61.7	77.6	90.0	98.6	112.6	123.6
Transforming	179.9	91.9	61.7	48.3	39.6	33.8	28.7	26.4	24.2	22.4
Bottom-up Evaluation	1024.2	522.1	331.8	254.9	204.5	169.2	142.7	129.1	113.5	104.6

Figure 5.4: Time Performance (2) for varying over SP



Elapsed Processing Times in Seconds									
SP	100	150	200	250	300	350	400	450	500
Total	199.3	262.0	318.5	386.4	403.9	453.3	577.8	586.7	714.2
TC	168.1	240.9	296.8	366.1	384.2	434.8	557.9	568.1	695.2
Partitions	123.6	194.6	247.6	309.9	333.8	378.9	493.8	499.4	618.8
Transforming	22.4	16.8	13.7	12.1	9.4	8.5	9.8	8.2	9.1

Figure 5.5: Time Performance (3) for varying over SP

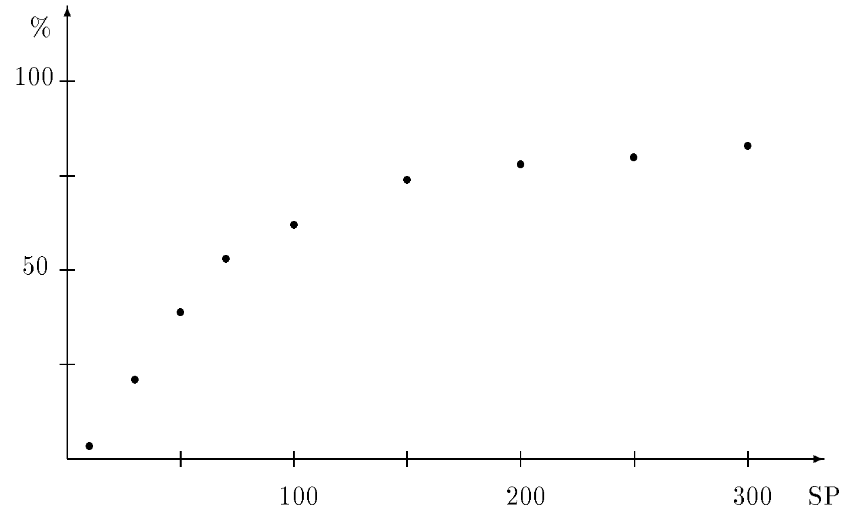


Figure 5.6: Proportion of time spent on computing the partitions Q_i

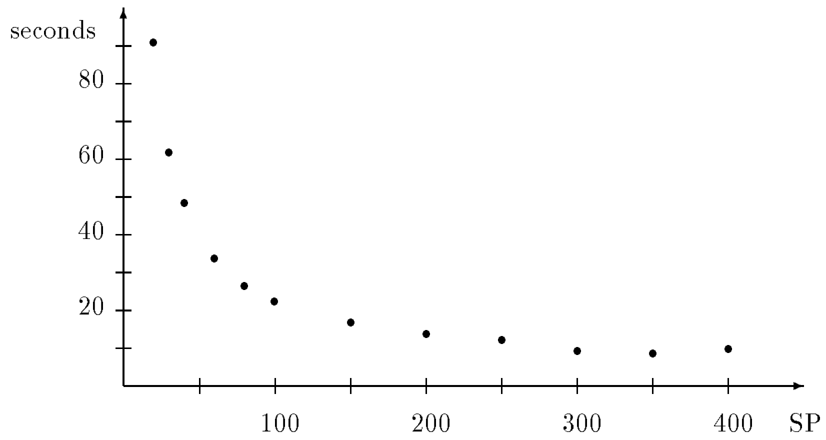


Figure 5.7: Time spent on transformations of a boolean matrix to a table representation of a relation and vice versa

variation over d means applying different queries to the same relation. The following family of queries was used.

$$p(X_0, X_1, X_2, \ldots, X_d, Z) \; :- \; e(X_0, X_1, X_2, \ldots, X_d, Z). \tag{5.1}$$

$$p(X_0, X_1, X_2, \ldots, X_d, Z) \; :- \; p(X_0, X_2, X_3, \ldots, X_d, X_1, Y),\, q(X_0, Y, Z). \tag{5.2}$$

$$p(X_0, X_1, X_2, \ldots, X_d, Z) \; :- \; p(X_0, X_1, X_2, \ldots, X_d, Y),\, q_d(X_0, Y, Z). \tag{5.3}$$

(5.1) is the exit rule, (5.2) is the linear recursive rule and (5.3) the d-step rule derived from (5.2). The relation Q_d is given by the nonrecursive rule

$$q_d(X_0, Y_d, Z) \; :- \; q(X_0, Y_d, Y_{d-1}),\, \ldots,\, q(X_0, Y_2, Y_1),\, q(X_0, Y_1, Z). \tag{5.4}$$

The relation Q is of the same family as the relations used in section 5.2.1. It has the parameters $S = 2000$, $SP = 50$ and $LP = 40$. The respective exit relation E is very similar: it holds 2000 tuples and the column referred to by Z includes the values of the 'Y-column' of Q. Each tuple of E holds the same value in each X_i attribute.

The effect of an increasing d on the performance can be seen in figure 5.9: due to the increasing time spent on computing the joins the total processing time grows. The time spent on computing the transitive closures decreases as these transitive closures are computed for the d-step relation Q_d which holds smaller longest paths for each partition as d increases. Regarding the graph representations of Q and Q_d (see figures 5.2 and 5.8) this becomes obvious: an edge in Q_d includes d edges of Q. This advantage has been obtained by computing Q_d by $(d-1)$ joins as can be seen by the Datalog equation for (5.4)

$$Q_d(X_0, Y_d, Z) = Q(X_0, Y_d, Y_{d-1}) \bowtie \ldots \bowtie Q(X_0, Y_2, Y_1) \bowtie Q(X_0, Y_1, Z)$$

We can summarize that the overall effect of having higher values for d is that the total processing time is more and more dominated by the behaviour of the join algorithm whereas the influence of the transitive closure component decreases.

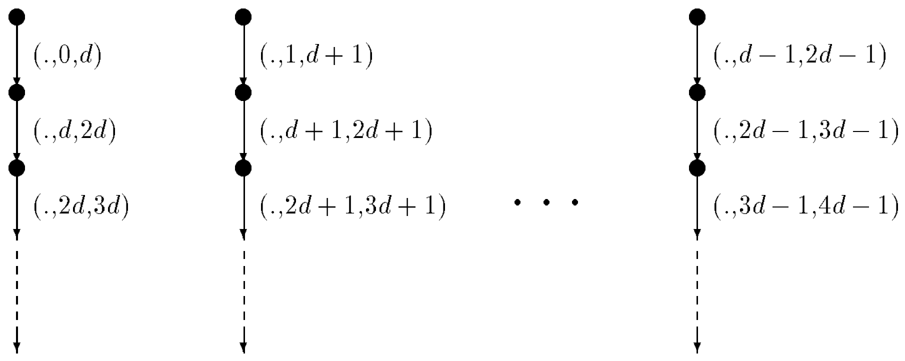
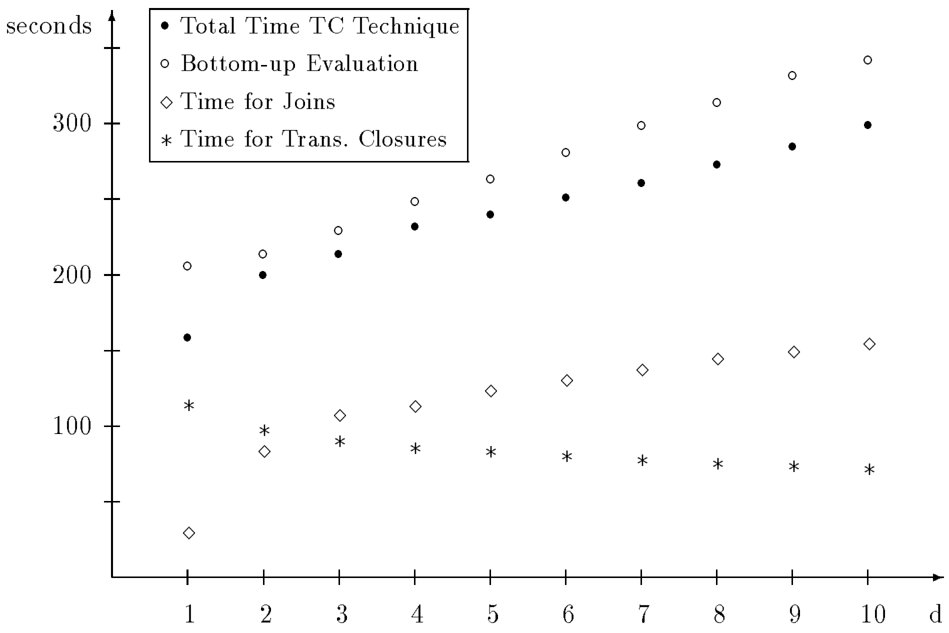


Figure 5.8: The graph representation of a **single** partition of Q_d



Elapsed Processing Times in Seconds

d	1	2	3	4	5	6	7	8	9	10
TC Technique	158.6	200.1	214.3	226.8	240.4	250.9	261.3	272.7	285.1	295.7
Joins	25.4	79.3	103.4	109.4	119.3	126.0	133.3	139.9	144.8	150.3
TC	114.8	97.8	91.0	86.7	83.8	81.2	78.4	76.2	74.5	72.3
Bottom-up	204.5	214.0	229.4	248.4	263.3	281.1	298.8	314.2	332.8	342.3

TC Technique = Total time using the transitive closure technique (includes join and TC times)
Joins = Time spent on computing joins
TC = Time spent on computing transitive closures
Bottom-up = Processing time for a bottom-up evaluation

Figure 5.9: Performance results for varying over d

Chapter 6

Conclusions

In this report we showed how linear recursive Datalog queries can be effectively processed. The two major issues were:

- replacing linear recursion by transitive closures and
- using data parallel algorithms.

The approach was to analyse the substitution graph of a linear recursive rule and to compute two parameters d and g from such a graph [Jag 87]. Finally the 'd-step rule' of the linear recursive rule can be derived. It is in the general basic form for transitive closure (3.2). 'd-step' refers to the fact that a bottom-up application of this d-step rule has the same effect as d successive bottom-up applications of the linear recursive rule from which it was derived. The advantage is that a computationally expensive bottom-up evaluation of the d-step rule can be replaced by computing a data-dependent number of transitive closures. Several data parallel transitive closure algorithms can be used. The choice of the most effective algorithm depends on the underlying graph structure of the database relation that is involved and on the actual number of available processors.

Finally we investigated the influence of two parameters on the performance of the processing technique:

- the number SP of partitions of the relation Q in the general basic form (3.2) and
- the diameter d of the substitution graph of the linear recursive rule.

SP depends on the actual data and may be obtained from the information stored in a database catalogue whereas the diameter d is induced only by the linear recursive rule of the query and is therefore independent from the actual data.

If SP increases, the total processing time becomes dominated by computing the partitions of Q. For high SP-values the proportion of time spent on this step was over 80%. In this case it is more effective to choose a naive bottom-up strategy to evaluate the query although this technique could benefit from the acyclic graph structure of the sample relations.

An increasing diameter, d, results in an increasing query processing time. For high d-values query processing using the transitive closure technique is dominated by the join algorithm because the number of joins is proportional to d. On the other hand the time spent on computing the transitive closures decreases because the underlying graph structures become smaller.

These results may be incorporated in an optimizing component which can choose the most effective evaluation technique. More specific strategies can be obtained by extending the analysis of the query. Substitution graphs only involve the two recursive literals (the consequent and the antecedent) of the linear recursive rule. A more detailed analysis can be obtained by using I-graphs which include information not only about the recursive but also about the nonrecursive literals of the linear recursive rule. This leads to more detailed and specific evaluation plans [You 92].

The class of queries can be extended by allowing nonlinear recursion. In this case the query transformation techniques of Saraiya and Zhang can be incorporated [Sar 89] [Zhang].

The combination of all these approaches may finally lead to a powerful optimizing component for Datalog queries.

Acknowledgements

This project was implemented at the Department of Computer Science of the University of Edinburgh and the Edinburgh Parallel Computing Centre which provided the equipment that was necessary to do this work.

I like to thank Peter Thanisch for discussing patiently every single problem with me, supporting and supplying me with copies of literature, handbooks etc., getting in touch with other people for further help, reading and correcting my reports and finally for that he was willing to supervise this project at all.

I like to thank also Malcolm Brown for giving me a lot of hints especially for the efficient implementation on the CM and for spending several hours on searching for another bug in the code.

Finally I thank Prof. P. C. Lockemann for his support to get this project accepted at the University of Karlsruhe which was a precondition to start working on it.

Bibliography

[Agr 88] R. Agrawal, H. V. Jagadish, *Multiprocessor Transitive Closure Algorithms*, Proceedings of the International Symposium on Databases in Parallel and Distributed Systems, Austin, Texas, 1988, pp. 56-66

[Agr 90] R. Agrawal, S. Dar, H. V. Jagadish, *Direct Transitive Closure Algorithms: Design and Performance Evaluation*, ACM Transactions on Database Systems, September 1990, pp. 427-458

[Akl 89] Selim Akl, *The Design and Analysis of Parallel Algorithms*, Prentice-Hall 1989

[Can 69] L. E. Cannon, *A Cellular Computer to Implement the Kalman Filter Algorithm*, Ph. D. Thesis, Montana State University, 1969

[CM 91] *Connection Machine CM-200 Series Technical Summary*, Thinking Machines Corporation, June 1991

[Gra 87] Gösta Grahne, Seppo Sippu, Eljas Soisalon-Soininen, *Efficient Evaluation for a Subset of Recursive Queries*, Proc. 6th ACM Symp. on Principles of Database Systems, 1987, pp. 284-293

[Ioa 85] Y. Ioannidis, *A Time Bound on the Materialization of Some Recursively Defined Views*, Proc. of the 11th International Conference on Very Large Databases, Stockholm, Sweden, Aug. 1985

[Jag 87] H. V. Jagadish, R. Agrawal, L. Ness, *A Study of Transitive Closure As a Recursion Mechanism*, Proc. ACM SIGMOD 1987 Conference on Management of Data, pp. 331-344

[Joh 87] S. L. Johnsson, *Communication Efficient Basic Linear Algebra Computations on Hypercube Architectures*, Journal of Parallel and Distributed Computing, April 1987, pp. 133-172

[Joh 89] S. L. Johnsson, Ching-Tien Ho, *Multiplication of Arbitrarily Shaped Matrices on Boolean Cubes Using the Full Communications Bandwith*, Yale University DCS Technical Report TR 721, 1989

[Nau 87] J. F. Naughton, *One-Sided Recursions*, Proc. 6th ACM Symp. on Principles of Database Systems, 1987, pp. 340-348

[Sag 88] Y. Sagiv, *Optimizing Datalog Programs*, in: Deductive Databases and Logic Programming, J. Minker, ed. Morgan-Kaufmann 1988

[Sar 89] Y. P. Saraiya, *Linearizing Nonlinear Recursions in Polynomial Time*, Proceedings of the 8th ACM Symposium on Principles of Database Systems, 1989, pp 182-189

[Sei 89] D. Seipel, *Decomposition of Linear Recursive Logic Programs*, Proc. International Workshop on Graph-Theoretic Concepts in Computer Science, Berlin, June 1990, pp. 291-310

[Shmueli] O. Shmueli, *Decidability and Expressiveness Aspects of Logic Queries*, ACM PODS, San Diego, CA, pp. 237-249

[Ull 88] J. D. Ullman, *Database and Knowledge-Base Systems*, volume 1, Computer Science Press, 1988

[Ull 89] J. D. Ullman, *Database and Knowledge-Base Systems*, volume 2, Computer Science Press, 1989

[Yan 90] M. Yannakakis, *Graph-Theoretic Methods in Database Theory*, Proceedings of the 9th ACM Symposium on Principles of Database Systems, 1990, pp. 230-242

[You 92] Ch. Youn, H.-J. Kim, L. J. Henschen, J. Han, *Classification and Compilation of Linear Recursive Queries in Deductive Databases* IEEE Transactions on Knowledge and Data Engineering, February 1992, pp. 52-67

[Zhang] W. Zhang et. al., *Necessary and Sufficient Conditions to Linearize Doubly Recursive Programs in Logic Databases* ACM Transactions on Database Systems, 15, 3, pp. 459-482

Index