Optimization Strategies for Parallel Linear Recursive Query Processing

Thomas Zurek Peter Thanisch

Department of Computer Science
Edinburgh University
Scotland
Email: {tz,pt}@dcs.ed.ac.uk

Technical Report ECS-CSG-16-95

July 1995

Abstract

Query optimization for sequential execution of non-recursive queries has reached a high level of sophistication in commercial DBMS. The successful application of *parallel* processing for the evaluation of *recursive* queries will require a query optimizer of comparable sophistication. The groundwork for creating this new breed of query optimizer will consist of a combination of theoretical insight and empirical investigation. Restricting our attention to *linear* recursive queries, we illustrate this process by developing a family of query processing strategies and, through experiments on a parallel computer, obtaining the basic information needed for an optimizer's heuristics.

1 Introduction

Query optimizers for commercial DBMSs are very sophisticated in their ability to find reasonably efficient execution plans for non-recursive queries computed on a uniprocessor. In such queries, the cost of performing the joins dominates the total cost and the optimizer must choose

(a) an order in which to execute the joins and

(b) for each join, an appropriate join technique (e.g. nested loop or sort/merge).

Two trends in database technology, namely parallel computing and the introduction of recursion into query languages, have made the task of query optimization much harder. Parallel computing has added an extra dimension to the optimization problem because the optimizer may have to decide on the computational resources to be allocated to each database operation and it may also have to decide on which *sets* of operations may be executed concurrently.

There are good theoretical and practical grounds for believing that recursive queries may be harder to optimize. Intuitively, a non-recursive SQL-query can be represented, for optimization purposes, as an algebraic expression with a fixed number of join operations, whereas a recursive query may, in general, require a data dependent number of join operations and/or additional operations, such as transitive closures, which may be even more computationally expensive than join operations.

Much research has been published on the subjects of

(a) exploiting parallel computing in query processing, e.g. in [KM91], [AJ88], [VK88b], and

(b) techniques and operators for recursive query processing, e.g. in [Agr87], [IW91], [JAN87], [Nau87], [NRSU89], [Sag88], [VB86], [VK88a], [YKLH92] and many others.

Furthermore some research has also examined the application of parallel computing to recursive query processing [CW89].

In order to bring this work into the mainstream of commercial DBMS query processing, it is necessary to develop an appropriate query optimizer. The difficulty of this class of optimization problem means that such an optimizer is likely to be driven by heuristics. The parameters for such heuristics may have to be obtained from observing the performance of the algorithmic technique on the target parallel computer. We illustrate this process in the case of *linear* recursive queries.

After some outlining comments about linear recursion – probably the most common class of recursion in queries [BR86] [JAN87] [YKLH92] – section 2 concludes with an example of a linear recursive query which is sufficiently general to show the relevant computational properties of two processing techniques, namely

- bottom-up evaluation (section 3) and
- transitive closure evaluation (section 4).

The transitive closure evaluation is used in our parallelization of the technique presented by Jagadish *et. al.* [JAN87]. These two techniques are used in section 5, where we describe a *generation matrix* model of query processing. In this model, the subcomputations required to process the query are laid out in the form of a gameboard and each query processing strategy corresponds to a particular traversal of this gameboard. The model provides a framework for analyzing processing strategies and for involving several efficiency issues such as parallelism and optimization features. The cost model presented in section 6 identifies the most important parameters that affect query processing performance or, in terms of the model, the costs of a move. In section 7 we give performance results obtained from an experimental implementation on a Connection Machine CM-200. Results are given for varying over three parameters:

- the length of the longest path in the underlying database relation (section 7.1),
- the number of transitive closures to be computed (section 7.2) and
- the data skew in the source and destination attributes (section 7.3).

Finally the paper is concluded in section 8.

2 Linear Recursion

In this paper, we are concerned with queries to relational databases where the user wants to interpret some of the data as a graph structure. In its simplest form, there will be a pair of columns in a table in which the entries are interpreted as the names of graph vertices and, for each tuple in the table, the sub-tuple corresponding to this pair of columns represents an arc in the graph.

Assume that we have a table, *Flight*, with four columns, *Airline*, *Flight*, *From* and *To*. Suppose that we want to know which cities are accessible from each city in the *From* column. Consider the following query[1]:

Example 1: *Accessibility.*

$$\begin{array}{rcl} C(X,Z) & = & Flight(A,F,X,Z) \\ C(X,Z) & = & C(X,Y) \bowtie Flight(A,F,Y,Z) \end{array}$$

This program is a representation of a simple graph connectivity problem which may be computed by a transitive closure operation. A well-known observation is that such a computation cannot be represented by an algebraic expression in which the operators are drawn from the traditional repertoire of relational algebra [AU79]. Of course, this observation is only important if one needs to manipulate algebraic expressions. A transitive closure computation could be performed by repeated joins, although the number of such join operations would be data-dependent.

If we give a logic programming style of representation to our queries, then we can say that a query is *linear recursive* if there is at most one subgoal of any rule that is mutually recursive with the head. Several authors have noted that linear recursive queries may be evaluated efficiently by using special-purpose evaluation techniques that cannot be applied to the more general class of recursive queries. Several such techniques are discussed in Ullman [Ull89].

Not all queries can be expressed as linear recursive queries. Consider the following example of a Datalog program.

Example 2: *Path system accessibility.*

$$\begin{array}{rcl} Access(Z) & = & Source(Z) \\ Access(Z) & = & Access(X) \bowtie Access(Y) \bowtie Triple(X,Y,Z) \end{array}$$

Here, the predicate names *Source* and *Triple* are assumed to correspond to tables stored in the database and the query computes the intensional relation *Access*. The *Path System Accessibility* problem is used by Afrati and Cosmadakis [AC89] as an example of a query that cannot be computed by any linear recursive Datalog program.

Thus linear recursive queries form an interesting class of queries that are strictly less expressive than the class of recursive queries in general, and yet which appear to me more expressive

[1]These are *Datalog equations* [Ull88]. Throughout this paper we rather prefer describing the queries by using relational algebra operators to make the reader aware of the operations that are involved in query processing although the resulting expressions are in 'Datalog style', i.e. we use Datalog-like notations like $P(X,Y,Z)$ and refer to attributes in the relation P by variables X,Y,Z. This also simplifies the notation of a join as the join conditions can be easily derived by matching the variables. Therefore we just denote a join by $\bowtie$ without giving the conditions explicitly. This notation is a hybrid of relational algebra and logic programming notation.
Using relational algebra expressions suggests that the techniques proposed in this paper can be applied to a wide range of query languages and not only to Datalog.

than the class of recursive queries that may be computed by a transitive closure. This last remark requires some qualification.

As has been noted by Jagadish *et al.* [JAN87] and Ullman [Ull89], we can compute a linear recursive query with a single transitive closure operation, but the underlying graph structure is *not* the one obtained simply by extracting the the 'from' and 'to' columns of the rows. The vertex set is potentially much larger as vertex names must be qualified with additional information; look at the following example.

Suppose that an awkward customer insists that for each flight in his/her journey, the airline must be the same, but it does not matter which airline is used. The following Datalog program provides the answer to this version of the accessibility problem.

Example 3: *Accessibility by the Same Airline.*

$$\begin{aligned} CA(A,X,Z) &= Flight(A,F,X,Z) \\ CA(A,X,Z) &= CA(A,X,Y) \bowtie Flight(A,F,Y,Z) \end{aligned}$$

The Datalog program in example 3 can also be computed using a transitive closure operator. However, this process is not as straightforward as in example 1. We have a choice between (a) executing one transitive closure operation for each airline and (b) constructing a much larger graph in which the names of the vertices are formed from a pair (A, X), where A is the name of an airline and X is the name of a city.

In this paper, we develop a family of parallel strategies for the sequential technique described in Jagadish *et al.* [JAN87]. This enables us to avoid the large graph structure described by Ullman [Ull89], replacing it with a collection of smaller transitive closure operations which may be computed in parallel.

Consider the linear recursive query in example 4, below. Sections 3 and 4 will then present two strategies for evaluating this query. Although such an example cannot claim to cover every aspect of query processing we rather prefer this way to give the reader the basic ideas behind the processing model that is presented in section 5 of the paper. Apart from that we can claim that the following example is sufficiently general to emphasize the important issues of linear recursive query processing as the linear recursive rule is in, what Jagadish *et al.* call, the *canonical form for transitive closure*. *Every* linear recursive rule can be transformed into this form. This form is essential for applying the transitive closure technique of section 4 but it is *not* necessary for the naive bottom-up evaluation presented in section 3.

Example 4:

$$P(X_1, X_2, Z) = E(X_1, X_2, Z) \tag{1}$$

$$P(X_1, X_2, Z) = P(X_1, X_2, Y) \bowtie, Q(X_1, Y, Z) \tag{2}$$

We will refer to (1) as the *exit rule* or *exit equation* and to (2) as the *linear recursive rule/equation* of the query.

For our purposes we can assume X_1, X_2, Y, Z to be single arguments but there is no problem to consider them as lists or ordered sets[2] of arguments.

[2]which implies that there are no repeated arguments. This assumption is no restriction because repeated arguments can be easily projected out from the canonical form and replicated after the essential query processing steps have been performed. The interested reader might refer to [JAN87].

3 Bottom-Up Evaluation

As is well known, a linear recursive query may be evaluated 'bottom-up', i.e. starting with the initial relation given by the exit rule further tuples are added by applying the linear recursive rule until no more new tuples are added. We will describe this process using the example 4.

For the purpose of a bottom-up evaluation the rules (1) and (2) are used in the following way:

1. The initial tuples are given by the exit rule (1). These tuples are said to be of *generation* P_0 so
$$P_0(X_1, X_2, Z) \;=\; E(X_1, X_2, Z)$$

2. Further tuples are added by sequential applications of equation (2) to produce the generations P_1, P_2, ... :
$$P_{i+1}(X_1, X_2, Z) \;=\; P_i(X_1, X_2, Y) \bowtie Q(X_1, Y, Z)$$

Therefore a bottom-up evaluation produces the generations P_i of tuples in a sequential order starting with the initial relation E as P_0 and joining Q in each step:

$$\begin{aligned}
P_0(X_1, X_2, Z) &= E(X_1, X_2, Z) \\
P_1(X_1, X_2, Z) &= E(X_1, X_2, Y) \bowtie Q(X_1, Y, Z) \\
P_2(X_1, X_2, Z) &= E(X_1, X_2, Y_1) \bowtie Q(X_1, Y_1, Y) \bowtie Q(X_1, Y, Z) \\
P_3(X_1, X_2, Z) &= E(X_1, X_2, Y_2) \bowtie Q(X_1, Y_2, Y_1) \bowtie Q(X_1, Y_1, Y) \bowtie Q(X_1, Y, Z) \\
& \ldots
\end{aligned}$$

Finally the resulting relation P of the query is obtained by

$$P = \bigcup_{i \geq 0} P_i = \bigcup_{i=0}^{M} P_i$$

such that

$$P_j \subseteq \bigcup_{i=0}^{M} P_i \;\text{ for all } j > M \tag{3}$$

which means that all P_j with $j > M$ do not contribute any new tuple to P. The existence of such an M is guaranteed [Ull88] [Ull89]. When we refer to M in this paper then M is considered to be the smallest number such that (3) holds.

The notion of representing the resulting relation as a set of generations of tuples is quite useful. Note that a generation is a set of tuples that were generated in the same way and generations do not have to be disjoint as some tuples might be derived in several ways. The union of all generations represents the result of the query. In figures in this paper, the set of tuples in a generation shall be represented graphically as a square.

A bottom-up evaluation can then be considered as producing a chain of generations, up to generation M, as in figure 1 where we get generation P_{i+1} by applying the linear recursive rule (or equation) to generation P_i.

We note that a bottom-up evaluation requires M join operations where M depends on the characteristics of the underlying data relations.

$$P_3(X_1, X_2, Z) = P_2(X_1, X_2, Z) \bowtie Q(X_1, Y, Z)$$

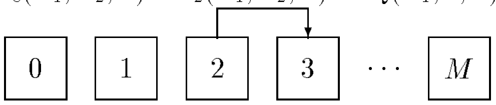

Figure 1: Graphical representation of a bottom-up evaluation producing a chain of tuple-generations.

4 Transitive Closure Evaluation

The example of section 3 shows that evaluating linear recursive queries can be done by computing a number of equi-joins involving the relation Q. If we look at the Datalog equations to compute a generation P_i we can see that the chain of joins on Q represents computing a kind of transitive closure:

$$\begin{aligned} P_i(X_1, X_2, Z) &= E(X_1, X_2, Y_i) \bowtie Q(X_1, Y_i, Y_{i-1}) \bowtie Q(X_1, Y_{i-1}, Y_{i-2}) \cdots \bowtie Q(X_1, Y_1, Z) \\ &= E(X_1, X_2, Y_i) \bowtie Q^i(X_1, Y_i, Z) \end{aligned}$$

with

$$Q^i(X_1, Y_i, Z) = Q(X_1, Y_i, Y_{i-1}) \bowtie Q(X_1, Y_{i-1}, Y_{i-2}) \bowtie Q(X_1, Y_1, Z)$$

In fact if we had all the Q^i's for $i = 1, \ldots, M$ we could easily compute all the P_i's:

$$\bigcup_{i=1}^{M} P_i = \bigcup_{i=1}^{M} E \bowtie Q^i = E \bowtie \bigcup_{i=1}^{M} Q^i$$

so the result, P, of the query could be computed by

$$P = \bigcup_{i=0}^{M} P_i = E \cup \bigcup_{i=1}^{M} P_i = E \cup (E \bowtie \bigcup_{i=1}^{M} Q^i)$$

We note that the union of the $Q^i(X, Y, Z)$ seems to be a transitive closure with Y as the source and Z the destination attribute. The problem is that there is also the X attribute so we can only combine tuples (X, Y, Z) with equal X-values. This results in transitive closures for every single value of attribute X: Let $\{a_1, \ldots, a_T\}$ be the values of the X attribute in $Q(X, Y, Z)$ and

$$Q_{a_j}(Y, Z) = \pi_{Y,Z}(\sigma_{X=a_j}(Q(X, Y, Z)))$$

The Q_{a_j}'s are the relations comprising the set of (Y, Z) subtuples of $Q(X, Y, Z)$ that are associated with the X-value a_j:

$$Q(X, Y, Z) = \bigcup_{j=1}^{T} (\{a_j\} \times Q_{a_j}(Y, Z))$$

So we can compute the union of the Q^i's by computing the transitive closures of the Q_{a_j}'s

$$\bigcup_{i \geq 1} Q^i = \bigcup_{j=1}^{T} (\{a_j\} \times Q_{a_j}^{+})$$

where $Q^+_{a_j}$ denotes the transitive closure of Q_{a_j}. Finally the result P of the query can be obtained by

$$P = P_0 \cup (E \bowtie \bigcup_{j=1}^{T} (\{a_j\} \times Q^+_{a_j}))$$

Essentially the processing technique shown in this example can be applied to *every* linear recursive query although there exist linear recursive queries that must be transformed into the canonical form of the query in the example so that the transitive closure technique can be applied [JAN87]. We avoid presenting the transformations and the processing algorithm in detail. The reader might refer to [JAN87] or [ZT93].

To show in which way the resulting relation P of a query is computed using the transitive closure technique we will present the essential steps and give a graphical representation of the method in section 5:

1. The substitution graph of the linear recursive rule of the query is analyzed and two characteristic parameters g and d are obtained. Appendix A gives two lemmas and a theorem [ZT93] to get the optimal values for g and d.
2. Let r be the linear recursive rule. We let r^d denote d applications of r and refer to r^d as the *d-step rule*. r^d is itself linear recursive. [JAN87] shows that a linear recursive query using generation P_g as the initial relation and having r^d as the linear recursive rule[3] can be processed in the way shown in the example.
3. $P_1, \ldots, P_g$ are computed by applying the linear recursive rule, r, g times to the initial relation $E = P_0$.
4. The d-step query involving r^d as the linear recursive rule and using P_g as the initial relation is evaluated by computing T transitive closures as shown in the example above. The resulting relation $\bar{P}$ of the d-step query is the union of the generations P_{g+id} (for all $i \geq 0$).
5. Finally the (original) linear recursive rule r is applied $(d-1)$ times to $\bar{P}$ to get the generations $P_{g+id+1}, \ldots, P_{g+(i+1)d-1}$ for $i \geq 0$.
6. The resulting relation P of the (original) query is the union of the generations computed in steps 3., 4. and 5.

5 The Generation Matrix Model

5.1 Generation Matrix

Figure 2 shows the generations P_i that form the result P of a linear recursive query. The matrix has d columns and a number of rows depending on the depth of the transitive closure[4]. Thus the number of rows is a data-dependent parameter whereas d is query-dependent.

Basically computing the result of a linear recursive query means computing the generations of its underlying generation matrix. We will now show graphically in which way these generations can be computed: we start in the top left corner as the generation P_0 (square 0) is given by

[3] We will refer to this query as the *d-step query*.

[4] i.e. the length of the longest path in the directed graph that is defined by the underlying relation.

the exit rule. Our goal is to proceed to every square (i.e. to compute every generation) in the matrix. Various operations are allowed that correspond to moves from one square to another:

(R) moving to the next generation to the right (i.e. computing P_{i+1} from P_i) means applying the linear recursive rule r to P_i (see figure 3),

(R2) moving two generations to the right (i.e. computing P_{i+2} from P_i) means applying the linear recursive rule r twice or applying the 2-step rule, r^2, once to P_i (see figure 4),

(Rd) moving to the generation below (i.e. computing P_{i+d} from P_i) means applying the linear recursive d-step rule, r^d, to P_i (see figure 5),

(TC) covering, or moving to, all generations of a column means applying the transitive closure technique using the d-step rule r^d. This move can only be executed if we have already reached or passed P_g (see figure 6).

Finally we can apply (R), (R2) and (Rd) simultaneously to a group of generations (squares) to reach their respective neighbours (see figure 7), i.e. applying a join to several generations, e.g.

$$P_{k+1} \cup P_{l+1} \;=\; (P_k \cup P_l) \bowtie Q$$

A summary of the moves is given in table 1; the costs of the moves are shown in table 2.

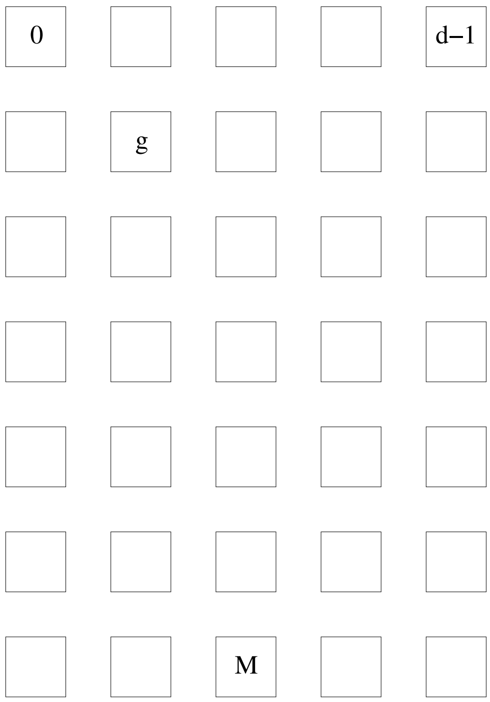

Figure 2: Generation matrix

Obviously we could also generate moves (R3) or (R4) to move three or four generations to the right by deriving the 3-step rule and the 4-step rule. These (Rx) moves do not appear to be of any practical use as they 'jump' over squares that have to be visited anyway but we should allow the general possibility of creating and applying such moves. Actually we will only use the (R) and the (TC) move.

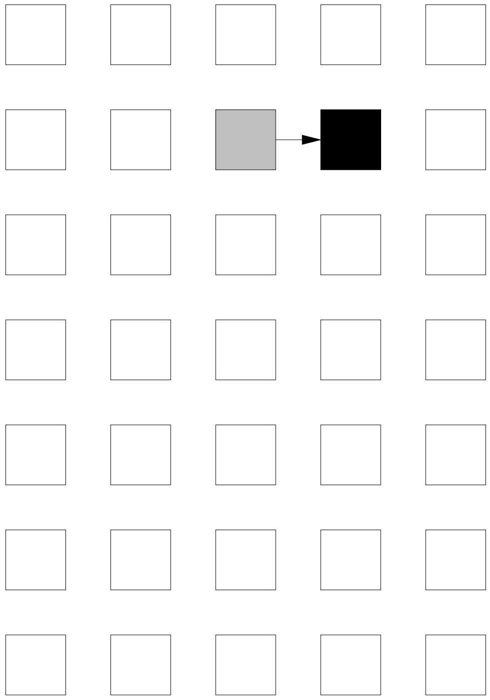

Figure 3: Moving one square to the right by applying the linear recursive rule r

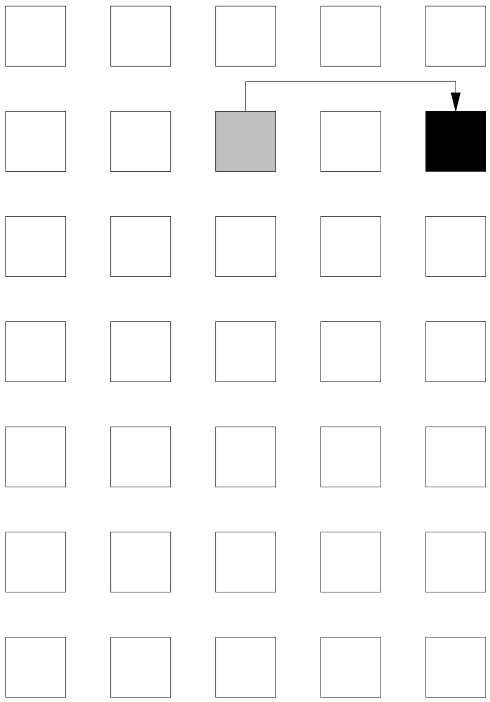

Figure 4: Moving two squares to the right by applying the 2-step rule r^2

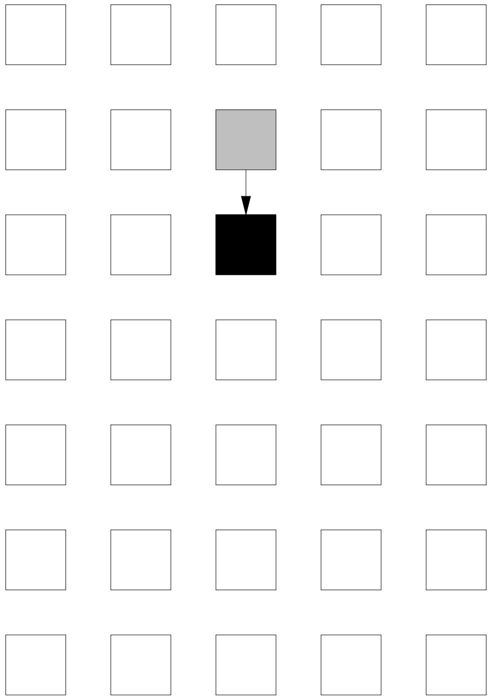

Figure 5: Moving one square down by applying the d-step rule r^d

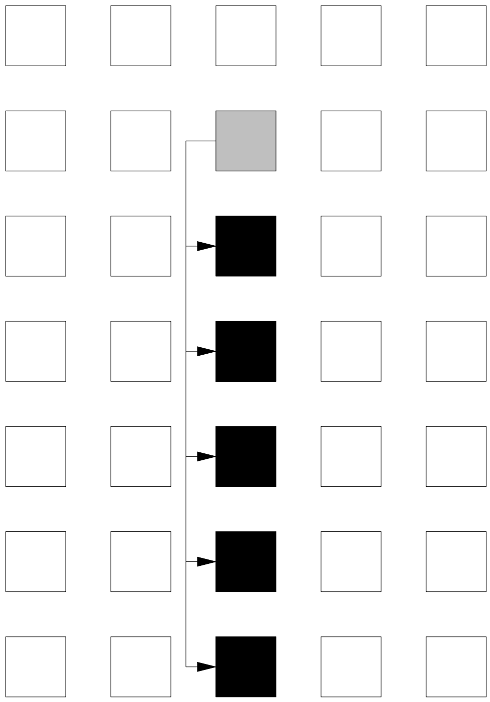

Figure 6: Covering the rest of a column by applying the transitive closure technique

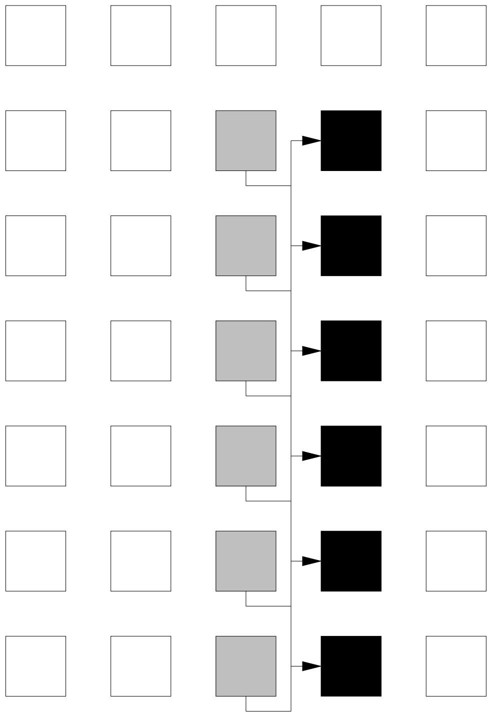

Figure 7: Simultaneous right move

move	corresponding operation
(R)	$P_{i+1} = P_i \bowtie Q$
(R2)	$P_{i+2} = P_i \bowtie Q \bowtie Q$
(Rd)	$P_{i+d} = P_i \bowtie \underbrace{Q \bowtie \cdots \bowtie Q}_{Q^d}$
(TC)	$Q^+_{a_j}$ for $j = 1, \ldots, T$

Table 1: Correspondence to operations

Several strategies/routes of 'running through the matrix' can be considered. The strategy described in section 4 (see steps 3., 4. and 5.) is the following; see also figure 8:

(a) Starting at square 0 perform right-moves (R) until you reach square g – this corresponds to step 3.

(b) Move down the column of square g – this corresponds to step 4 or a (TC) move.

(c) Perform $(d-1)$ right moves (R) from all squares of the respective column of square g simultaneously to get to the remaining squares. – this corresponds to step 5.

The simple bottom-up strategy is shown in figure 9.

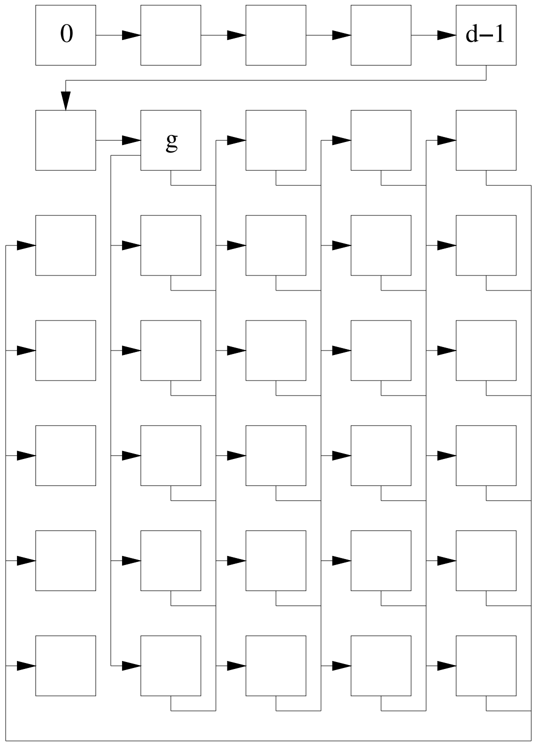

Figure 8: Transitive closure strategy of moving through the generation matrix

5.2 An Example

We will now give an example of how the transitive closure technique works. It will be used to give an idea of the algorithm proposed by [JAN87] and to avoid going into too many details.

The following query will be used throughout the example:

$$P(A,B,C,D,Z) = E(A,B,C,D,Z) \quad (4)$$
$$P(A,B,C,D,Z) = P(D,A,B,C,Y) \bowtie Q(Y,Z) \quad (5)$$

The substitution graph of this query is very simple; it is shown in figure 10. There is only one cycle of length 4 and by applying the lemmas and the theorem of appendix A we get $d = 4$ and $g = 0$.

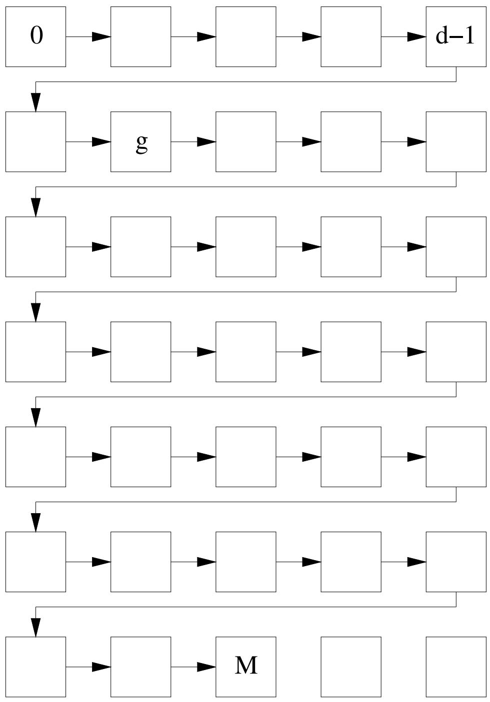

Figure 9: Bottom-up evaluation of a query

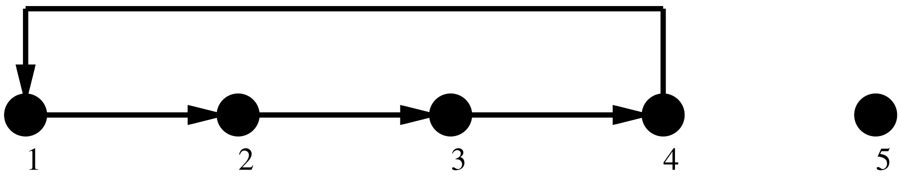

Figure 10: Substitution graph of the linear recursive rule

The following relations E and Q are used in the example:

$$E = \{(0,1,1,0,2),(1,0,1,0,3),(1,1,0,0,2),(0,0,1,1,2)\} \quad (6)$$
$$Q = \{(1,2),(2,3),(3,4),\ \ldots\ ,(15,16)\} \quad (7)$$

The first task is to derive the d-step rule r^d which is in this case the 4-step rule r^4. We compute r^4 by deriving r^2, r^3 and finally r^4:

$$\begin{aligned}
P(A,B,C,D,Z) &= P(D,A,B,C,Y_1) \bowtie Q(Y_1,Z) \\
P(A,B,C,D,Z) &= P(C,D,A,B,Y_2) \bowtie Q(Y_2,Y_1) \bowtie Q(Y_1,Z) \\
P(A,B,C,D,Z) &= P(B,C,D,A,Y_3) \bowtie Q(Y_3,Y_2) \bowtie Q(Y_2,Y_1) \bowtie Q(Y_1,Z) \\
P(A,B,C,D,Z) &= P(A,B,C,D,Y_4) \bowtie Q(Y_4,Y_3) \bowtie Q(Y_3,Y_2) \bowtie Q(Y_2,Y_1) \bowtie Q(Y_1,Z)
\end{aligned}$$

The last rule, r^4, can be reduced by creating a new relation Q^4:

$$\begin{aligned}
Q^4(Y,Z) &= Q(Y,Y_3) \bowtie Q(Y_3,Y_2) \bowtie Q(Y_2,Y_1) \bowtie Q(Y_1,Z) \\
P(A,B,C,D,Z) &= P(A,B,C,D,Y) \bowtie Q^4(Y,Z) \quad (8)
\end{aligned}$$

The relation Q^4 looks like this:

$$Q^4 = \{(1,5),(2,6),(3,7),\ \ldots\ ,(12,16)\} \quad (9)$$

Comparing Q and Q^4 shows why r^4 does in fact 4 steps and comparing the linear recursive rule r and its derivatives r^2 and r^3 with the 4-step-rule r^4 shows the significant characteristic of r^4: the argument patterns of the P-literals are *isomorphic*[5] whereas the respective patterns of the P-literals in r, r^2 and r^3 are not. In fact the isomorphism of the P-literals is the main characteristic of r^4 that allows us to apply the transitive closure technique with $P_g = P_0 = E$ as the exit relation.

Finally we have to compute one single transitive closure as Q^4 consists only of a source and a destination attribute and there is no attribute like X_1 in the example of sections 3 and 4. The transitive closure Q^{4+} of Q^4 is

$$\begin{aligned}
Q^{4+} = \{&(1,5),(1,9),(1,13),(2,6),(2,10),(2,14),(3,7),(3,11),(3,15), \\
&(4,8),(4,12),(4,16),(5,9),(5,10),\ldots,(12,16)\}
\end{aligned}$$

Now we have got several rules and relations to process the original query:

- the exit rule (4) of the query and the corresponding exit relation E (6),
- the linear recursive rule r (5) and the corresponding relation Q (7),
- the linear recursive 4-step rule r^4 (8) and the corresponding relation Q^4 (9).

Figure 11 shows the generations $P_0, P_1, \ldots, P_{15}$ that form as a union the resulting relation P. We will now show the structure of P by moving from one generation to another using the rules and relations given or derived:

[5] see appendix A for a definition of isomorphism.

- The tuples of the top left square or generation are given by the exit rule (4) so $P_0 = E$ and the tuples of E are put into the top left square.
- Note that the tuples of a generation P_{i+1} can be derived by applying (5) to the tuples of P_i, e.g. for $i = 0$:

$$\begin{array}{rcl} P_1(1,1,0,0,3) & = & P_0(0,1,1,0,2) \bowtie Q(2,3) \\ P_1(0,1,0,1,4) & = & P_0(1,0,1,0,3) \bowtie Q(3,4) \\ & \ldots & \end{array}$$

  This corresponds to an (R) move.
- Tuples of a generation P_{i+d} can be derived by applying (8) to the tuples of P_i, e.g. for $i = 0$:

$$\begin{array}{rcl} P_4(0,1,1,0,6) & = & P_0(0,1,1,0,2) \bowtie Q^4(2,6) \\ P_4(1,0,1,0,7) & = & P_0(1,0,1,0,3) \bowtie Q^4(3,7) \\ & \ldots & \end{array}$$

  This corresponds to an (Rd) move.
- As $P_g = P_0$ a (TC) move can be executed right from the beginning, i.e. we can compute

$$\bigcup_{k \geq 1} P_{i+kd}$$

  by applying the TC technique to a generation P_i, i.e. we can compute the remaining generations of the column of P_i by computing Q^{4+} – which we have already done – and joining the these tuples with those of P_i. We demonstrate this by using only the tuple $(0,1,1,0,2)$ of generation P_0:

$$\begin{array}{rl} & \{(0,1,1,0,2)\} \bowtie Q^{4+} \\ = & \{(0,1,1,0,2)\} \bowtie \{\ldots,(2,6),(2,10),(2,14),\ldots\} \\ = & \{(0,1,1,0,6),(0,1,1,0,10),(0,1,1,0,14)\} \end{array}$$

  These are exactly the tuples appearing in the first positions of generations P_4, P_8, P_{12} (see figure 11).
- (R2) or (R3) moves can be performed by applying the rules r^2 and r^3.

5.3 Parallel Aspects

The generation matrix model of section 5.1 provides a framework for analyzing several strategies to process a linear recursive query. The moves (R) and (TC) may be combined in several ways for query evaluation. Considerations of sequential strategies dwell upon whether to apply the (TC) move or to go rather for an ordinary bottom-up evaluation, i.e. applying (R) moves. This is a crucial point and we can open the ground for a wide range of new processing strategies by involving parallelism. Parallel issues in the evaluation process are:

Data parallel moves: Figure 7 gives already an example in which an (R) move can be applied to several portions of data (e.g. several generations) in parallel. This is called data parallelism as the same instructions are executed on different items of data. In this case

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

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

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

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

Figure 11: The generations in form of a matrix

data parallelism is possible because there are no constraints in between the portions of data. E.g. generations do not have to be disjoint and can be built independently from one another. This means that (R) moves – and the same applies to (TC) moves – can be performed simultaneously on several generations, e.g. as shown by figure 7.

Different moves in parallel: The (R) and (TC) moves do not interfere with each other so (R) and (TC) moves can be executed in parallel even on the same portion of data, i.e. (R) or (TC) can be applied whenever possible and regardless of whether there is already another activity going on.

Parallelizing the (R) move: Clearly the (R) move can be parallelized internally. As we have already seen it consists mainly of a join operation[6]. Therefore efforts should be mainly concentrated on implementing one or more appropriate[7] parallel join algorithms.

Parallelizing the (TC) move: The same ideas that have applied to the (R) move can be used for the (TC) move. The dominant operation in this case is clearly the transitive closure. There are several well investigated parallel transitive closure algorithms based on boolean matrix multiplications [Akl89], the Warshall constraints [ADJ90] or hash-join techniques [ZZO93]. Furthermore there are techniques regarding the characteristics of parallel computers, e.g. [AJ88], [CCH90], [DR94].

[6] Projection and selection operations may also be involved but can be neglected regarding the performance issues that are dominated by the join operation

[7] E.g. regarding the underlying machine's architecture.

6 The Cost Model

Usually a database management system (DBMS) incorporates a query optimizing module that decides on the best way to process the query. Given the generation matrix model and the moves and the parallel issues of sections 5.1 and 5.3, a variety of strategies to move through the generation matrix, i.e. of processing a query, can be considered. Naturally it would be nice to find out a way to determine the optimal, i.e. the fastest or cheapest, strategy in advance.

We can consider a strategy to be a sequence of stages that are consecutively executed. In the simplest, i.e. sequential, case a stage consists of a single move; in the more complex, parallel case a stage consists of several moves that are executed in parallel.

The costs of a strategy are the sum of the costs of all its stages and the costs of a stage are the costs of the most expensive move in a stage. The problem of calculating the strategy costs is therefore reduced to the problem of identifying the move costs.

Although the dimensions of the matrix and the values of d and g are parameters that influence the performance we may neglect them as they are the same for each of the competing strategies to process the query so these parameters are not of any use to distinguish between the strategies. We can therefore concentrate on the costs of the moves.

Table 2 shows that the costs of a move are not constant and depend parameters like T or on the size of the relations involved in a join operation. Therefore predicting the costs of a strategy is not straightforward. Basically we can concentrate on the following issues:

1. on the efficiency of a right-move (R), i.e. essentially the efficiency of the underlying join operations as one right-move is in fact executing one join operation, which is dominated by
   (a) the size of the databases relations involved in the join and
   (b) the data skew, i.e. whether the data values are distributed uniformly or not[8]
2. on the efficiency of the (TC) move; this is dictated by
   (a) the number T of transitive closures[9] for computing the generations of the column in which square g is located,
   (b) the sizes of the underlying graph structures and
   (c) the lengths of the longest paths in the graphs.

The sizes of intermediate join relations and the sizes of the underlying graph structures cannot be predicted easily [LN89]. Therefore we will follow a heuristic approach. The following section will present the costs for (R) and (TC) moves obtained by experiments on a Connection Machine CM-200. We give these results to prove the dependency of these costs on the lengths of the longest paths, on the number T of transitive closures and on data skew and to show in which way a query optimizer can use these parameters to choose the most appropriate strategy.

[8]There is a variety of papers covering the problems data skew causes on parallel joins, e.g. [WD91] [WDY93] [KO90].

[9]Actually T is also a skew parameter as it immediately depends on the number of values appearing in a certain attribute or set of attributes.

move	costs
(R)	1 join for each move
(R2)	1 join for each move 1 join for building $Q \bowtie Q$ (once)
(Rd)	1 join for each move $(d-1)$ joins for building Q^d (once)
(TC)	T selection operations to build the Q_{a_j}'s T transitive closures

Table 2: Costs of the moves

7 Move Costs

We now present the performance results for (R) and (TC) moves. These results were obtained from several sequences of experiments run on a Connection Machine CM-200. The CM-200 has an SIMD architecture using 16384 processors. It is an inherently data parallel machine which allows us to exploit all but the second of the parallel issues mentioned in section 5.3.

The dependency of the move costs on three parameters are shown:

- the length L of the longest path in the underlying graph,
- the number T of values occurring in the attribute(s) that are common to every literal in a linear recursive rule[10],
- the data skew amongst the source and destination attributes of the transitive closure(s).

The bottom-up and the transitive closure strategies were implemented using a data parallel hash join [Min93] [KO90] [KM91] and a simple join algorithm [ZT93]. The hash join spreads the data over a data-dependent number of processors which makes it sensitive to data skew effects (see sections 7.2 and 7.3) whereas the simple join uses the entire set of processors and essentially performs parallel selections to get the tuples that qualify for the join and builds the join result sequentially.

For computing the transitive closure an algorithm was implemented that is based on performing parallel boolean matrix multiplications [Akl89]. This proved to be the most efficient [ZT93].

The times given in the experiments are CM elapsed times in seconds and incorporate the times for interaction with the secondary storage.

7.1 Varying L

The query used in these experiments is the most simple for computing the transitive closure $P(X,Z)$ of a relation $Q(X,Z)$:

$$\begin{aligned} P(X,Z) &= Q(X,Z) \\ P(X,Z) &= P(X,Y) \bowtie Q(Y,Z) \end{aligned}$$

[10]see for example the X_1 attribute in the linear recursive rule (2).

The underlying graph structure[11] of Q was a number of binary trees each of them having a depth not larger than L. L was varied over the range of $\{2, 3, \ldots, 10\}$. The size of Q was 1000 tuples in each experiment. Thes results for performing one (R) and one (TC) move are shown in figures 12 and 13 respectively.

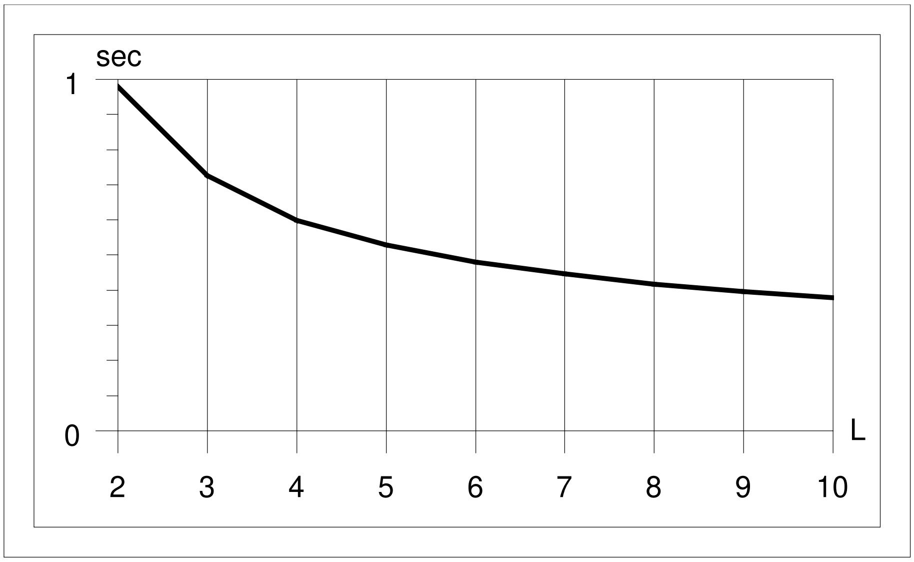

Figure 12: Costs of one (R) move depending on the length L of the longest path; a hash join algorithm was used.

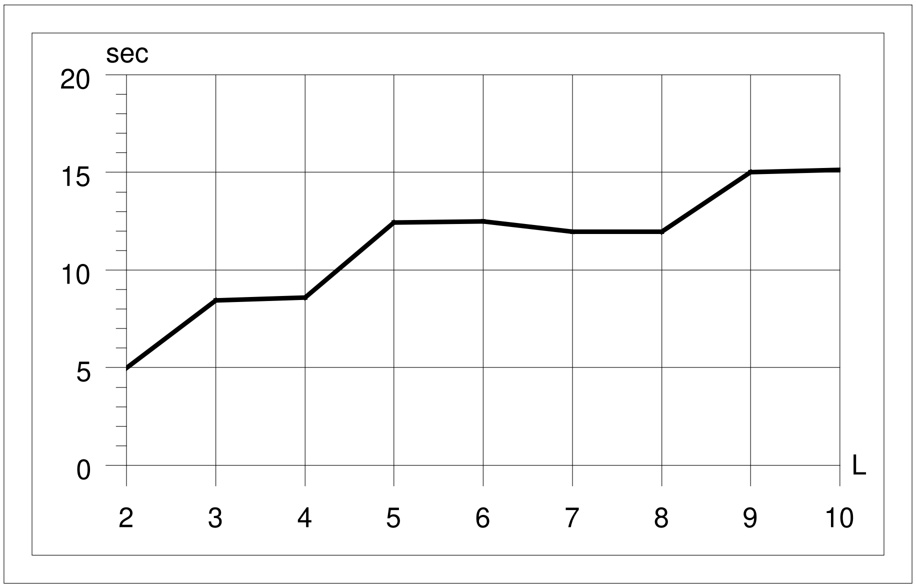

Figure 13: Costs of a (TC) move depending on the length L of the longest path.

Figure 12 shows that the average costs for an (R) move are cheaper for higher values of L. This is due to the fact that for low L values the average join result is larger than for high L

[11] A relation $Q(X, Z)$ can be considered as the set of edges of the graph, i.e. a tuple $(a, b) \in Q$ denotes that there is an edge from node a to node b. The values for attributes X and Z define the set of nodes of the graph.

values. This is reasonable as small L values force the data to be 'denser' i.e. to be connected by short paths. Therefore a larger number of nodes can be reached by combining two pathes (i.e. computing a join) that are already known.

The costs of a (TC) move are clearly dominated by the number of matrix multiplications to be performed. This number is given by $\lceil \log_2 L \rceil$ which causes the 'stair effect' in figure 13.

Figure 14 shows the overall performance of the two strategies in the experiments. The transitive closure performs better than the bottom-up strategy for higher values of L as it increases logarithmically with a linearly increasing L whereas the bottom-up strategy grows linearly. In figure 14 the crossoverpoint of the two graphs is around $L = 7$. This value depends also on implementation and hardware specific characteristics and should not be considered as a 'natural law' but the existence of such a crossover is the important fact. The corresponding value of L can be obtained experimentally in the specific case. On knowing this value an optimizer can choose the most effective strategy for processing a query.

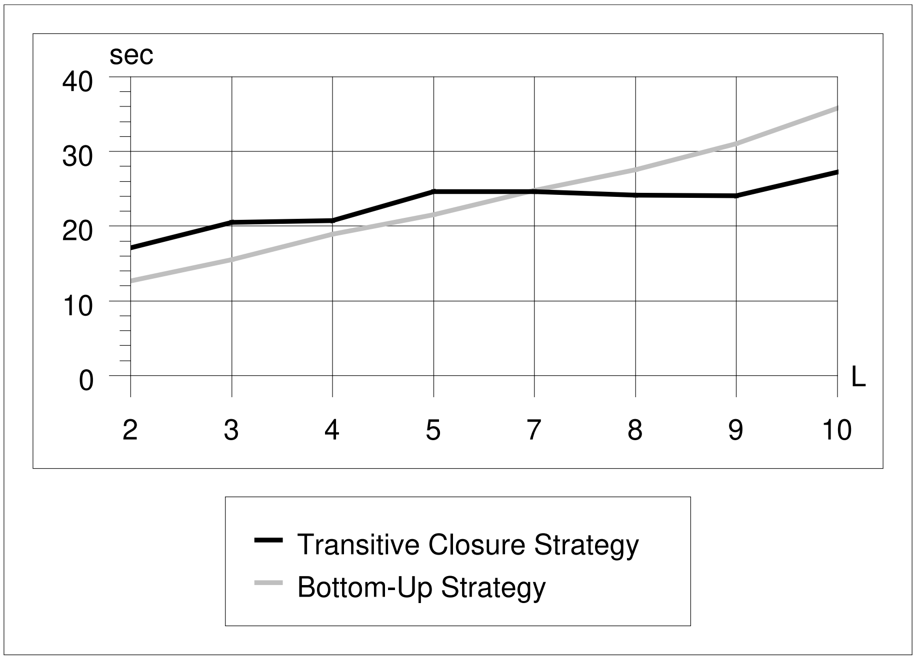

Figure 14: Overall performance of the strategies depending on L

7.2 Varying T

In these experiments the query (1), (2) of section 2 was used:

$$
\begin{aligned}
P(X_1, X_2, Z) &= Q(X_1, X_2, Z) \\
P(X_1, X_2, Z) &= P(X_1, X_2, Y) \bowtie Q(X_1, Y, Z)
\end{aligned}
$$

The underlying graph structure of Q were a number of simple pathes each one not longer than 10, i.e. L was set to 10 in each experiment. The number T of values of the attribute X_1 in $Q(X_1, X_2, Z)$ was varied over the range of $\{10, 20, \ldots, 100\}$; the size of Q was 1000 tuples.

Actually the parameter T characterizes a kind of redistribution skew of the X_1 attribute [WDJ91] which is relevant for parallel implementations of relational algebra operators. They often use hash functions depending on the data values of a particular attribute to spread the data over a number of processors. This implies that an increasing number of distinct values results in the data being distributed over a larger number of processors. This applies for the hash join algorithm we used in the experiments of figure 15; additionally the results for the simple parallel join are given. The latter algorithm is not sensitive to effects of redistribution skew.

Figure 15 shows the costs of a single (R) move using the hash join and the simple join respectively. The hash join is influenced by the redistribution skew and performs worse than the simple join for small values of T. Both join algorithms suffer from linearly increasing join results for T being increased but the hash join overcompensates this effect by achieving a better data distribution for large T values.

Figure 16 gives the costs for a (TC) move. The costs rise nearly linearly with increasing T values. This is due to the fact that T is the number of transitive closures that have to be computed in each case. The costs of each transitive closure is nearly constant although the underlying graph is larger for a smaller T. Nevertheless they remain constant in this case due to specific Connection Machine programming characteristics.

Finally figure 17 shows the overall performance of the transitive closure and the bottom-up strategies. As in the case of L in section 7.1 the remarkable fact is the existence of a crossover at $T \approx 43$. Again this T value depends on implementation and architectural characteristics and must be obtained experimentally. But the experiments prove again that it is worth to have an optimizer to decide on the most efficient strategy (including a decision on the most appropriate join algorithm).

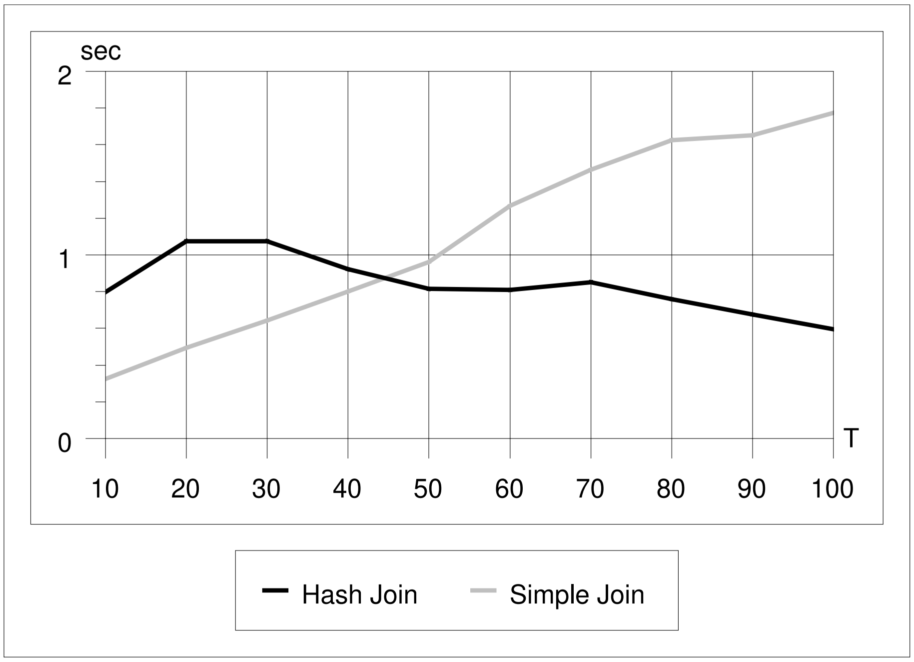

Figure 15: Costs of one (R) move depending on the number T of values in the X_1 attribute of $Q(X_1, X_2, Z)$ for a parallel hash join and a simple data parallel join.

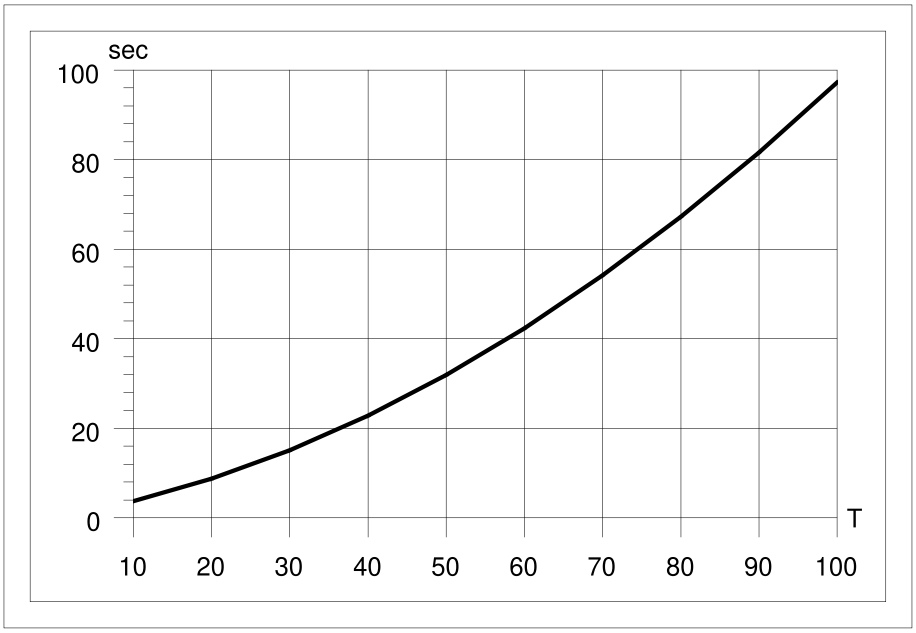

Figure 16: Costs of one (TC) move depending on the number T of values in the X_1 attribute of $Q(X_1, X_2, Z)$.

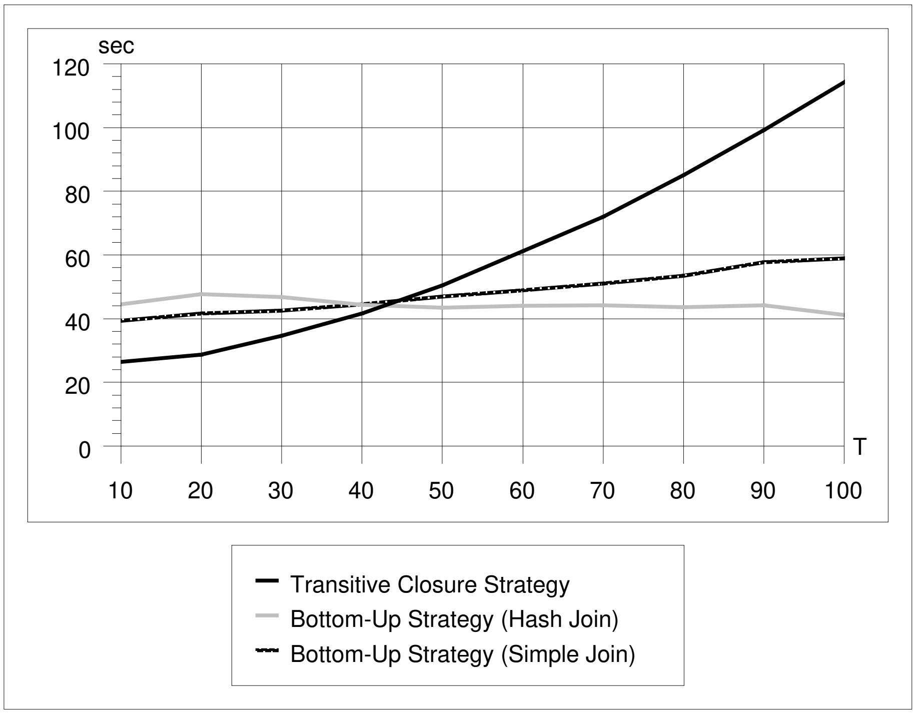

Figure 17: Overall performance of the strategies depending on T

7.3 Varying the Data Skew

The query used in these experiments is the query of section 7.1 was used for computing the transitive closure of a relation $Q(X, Z)$:

$$\begin{aligned} P(X,Z) &= Q(X,Z) \\ P(X,Z) &= P(X,Y) \bowtie Q(Y,Z) \end{aligned}$$

The underlying graph structure of Q was a number of DAGs (acyclic directed graphs). The size of $Q(X, Y)$ was set to 1000 tuples and each of the attributes held 333 distinct values 0,1,. . .,332 so redistribution skew was fixed. The values of the X and Y attributes were Zipf-distributed [Knu73] with

$$x_i = \frac{c}{i^\theta}$$

to be the probability of value i in the X attribute where

$$c = \frac{1}{H_{333}^{(\theta)}} \quad \text{and} \quad H_{333}^{(\theta)} = \sum_{k=1}^{333} \frac{1}{k^\theta}$$

So in these experiments the tuple placement skew [WDJ91] was varied rather than the redistribution skew, i.e. we varied over the frequency of the values rather than the number of distinct values.

The probability y_i of value i in the Y attribute was set to

$$y_i = x_{333-i-1}$$

which results in contrary distribution as shown in figure 18. Having the values distributed in this way allows to generate a sequence of relations Q synthetically such that

$$x < y \ \text{ for every } \ (x, y) \in Q(X, Y)$$

which guarantees acyclicity. The resulting DAGs had different lengths of their respective longest paths so the depth of the recursion was fixed to 4 in all experiments to avoid the results being influenced by that parameter.

Figure also shows the effect of varying the parameter θ: a small θ value means less skewed data and results in an increased overlap of the values which implies larger join results for $Q(X, Y) \bowtie Q(Y, Z)$ – see the diagrams in figure 18.

Figure 19 presents the costs for an (R) move depending on the skew parameter θ. The hash join benefits from low θ values as they imply a better distribution amongst the 333 processors that are involved in the computation. Both join algorithms benefit from decreasing join results for an increasing data skew. This is due to the effect shown in figure 18 that the overlap decreases with in a θ increasing.

The result shown in figure 20 show that the (TC) move is not affected by data skew. This is caused by the fact that the boolean matrix multiplication technique was used for computing the transitive closure. There are several other parallel transitive closure algorithms which are based on hash techniques, e.g. [ZZO93], and which are therefore sensitive to skew effects.

Figure 21 gives the overall performance results for the strategies and summarizes the issues that were already discussed. [WDJ91] denotes changing join result sizes as another type of skew, namely the join product skew. Following this interpretation and stating that every join algorithm naturally depends on the size of the join result we can conclude from figure 21 that the bottom-up strategy is very sensitive to skew effects (either tuple placement or join product skew) whereas the transitive closure strategy can be implemented such that is nearly[12] unaffected by data skew.

8 Conclusions

After outlining the problems of linear recursion in section 2 we presented *two* evaluation techniques in sections 3 and 4 which were integrated in *one* query processing model in section 5. The generation matrix model is suitable as an abstract representation for a query optimizer because it provides a small set of simple operators which can be combined to create several strategies to process a query.

It also focuses attention computer resource utilization, as linear recursive queries have to be processed by computing a *data-dependent* number of transitive closures. This data-dependency *cannot* be expressed in relational algebra just by adding a transitive closure operator[13].

Thus a query optimization module in a database system that executes recursive queries on a parallel computer would operate in a way that is similar to a conventional query optimizer. A utility program would collect statistical information about the current state of the data. Certain properties, such as data skew, might be expected to change relatively slowly so the measurement of skew would hold good for some period of time. Various parameters for heuristics would be properties of the target parallel computer, and such parameters must be computed by experimentation.

In section 7 performance results were given for an implementation of the processing model on a data parallel architecture. The results prove that it is worth considering parameters like the lengths of the longest paths, the number of transitive closures and data skew for choosing the appropriate strategy for processing a particular query. Information about these parameters can be stored in the database catalog and are available before a query is processed.

[12] Naturally transforming the boolean matrix into the final result – a database relation – is also influenced by the number of resulting tuples that have to be generated. Figure 21 proves that this effect is not significant.

[13] By contrast we remind the reader that – in theory – every linear recursion can be replaced by computing the transitive closure of one, possibly very large graph [Ull89] [JAN87]. This can be expressed in a relational algebra enhanced by a transitive closure operator.

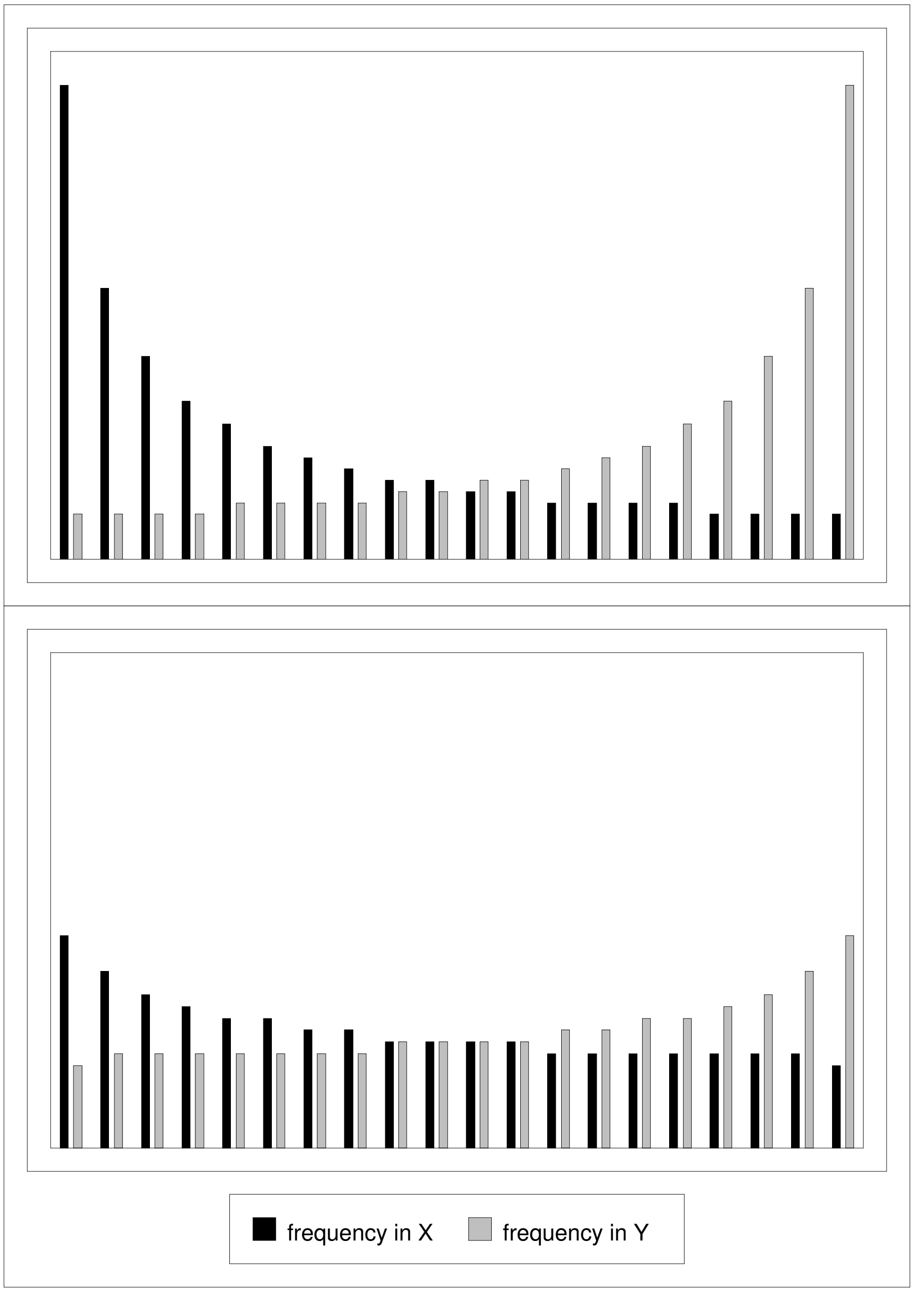

Figure 18: Zipf distribution of the values in the relations $Q(X, Y)$ that were used in the experiments. Each pair of columns shows the frequency of a particular value in the X and the Y attribute respectively. The upper diagram shows a distribution for a high θ value the lower for a small θ.

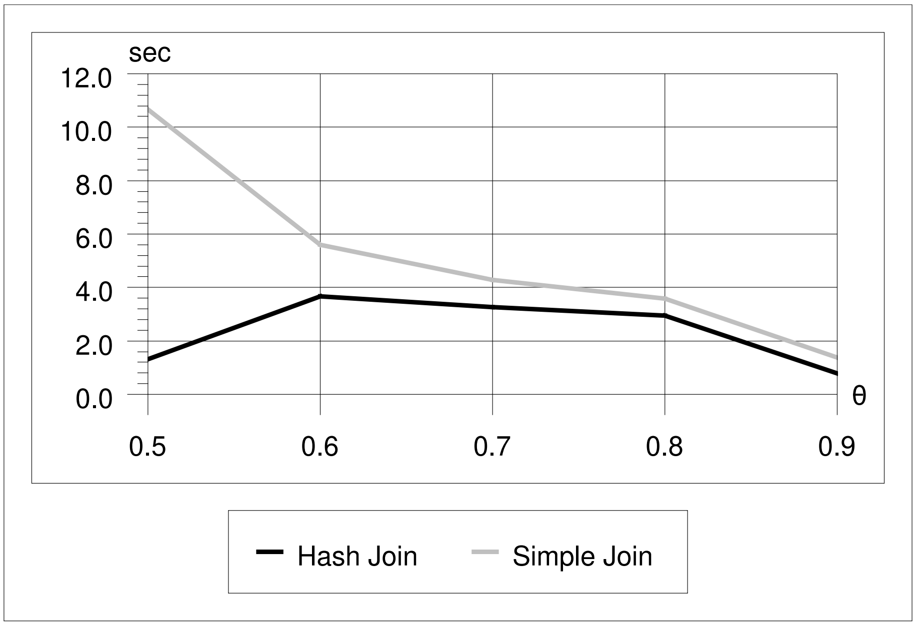

Figure 19: Costs of one (R) move depending on the skew parameter θ for the parallel hash join and a simple data parallel join.

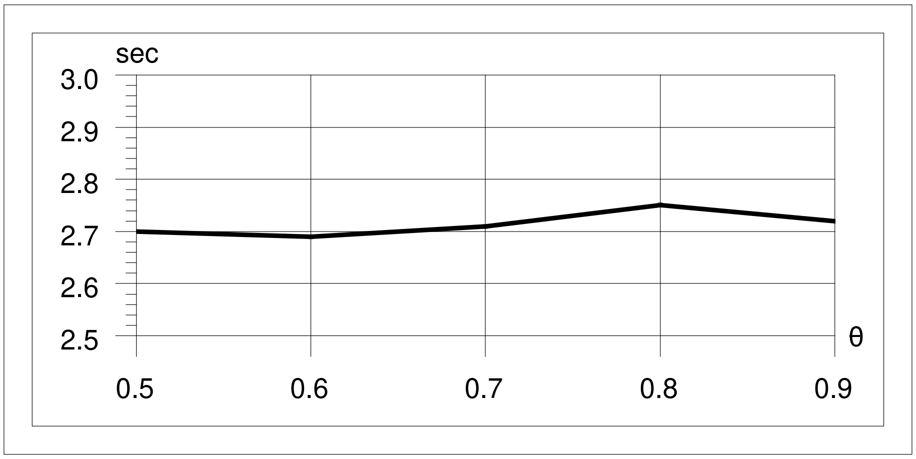

Figure 20: Costs of a (TC) move depending on the skew parameter θ.

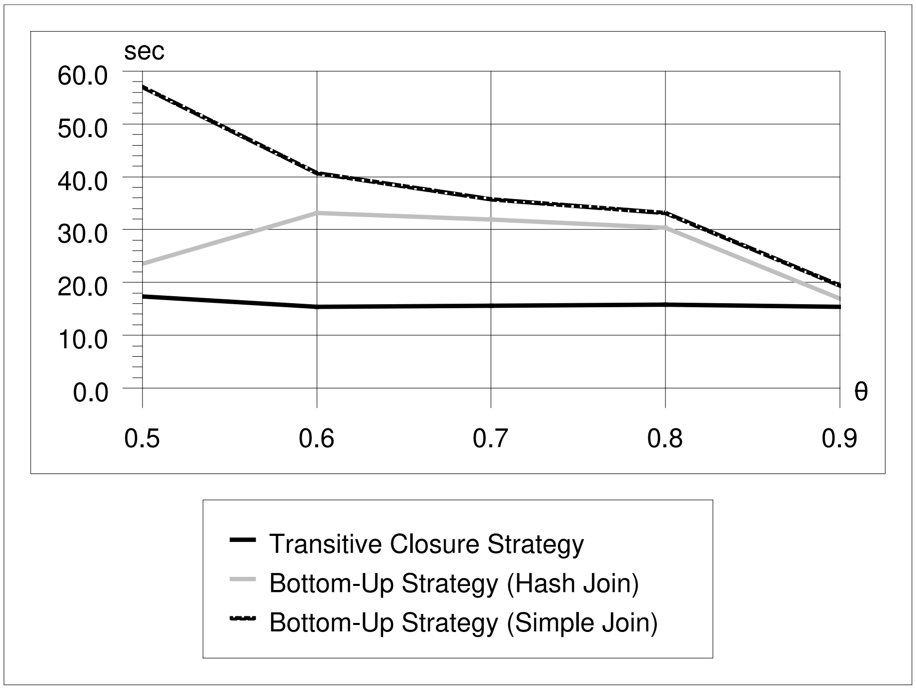

Figure 21: Overall performance of the strategies depending on θ

A Deriving Parameters d and g

In this section we will present the essential tools to derive the parameters d and g from a linear recursive rule. The lemmas and the theorem produces more precise values than given in [JAN87] which concentrates on the properties of d and g rather than the exact values.

Definition 1 (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 2 (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 *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

$$S((G, H)) = (H, I)$$

where I is a new variable.

In the following proofs we will label the vertices of the substitution graphs with the respective argument of the linear recursive rule. Applying a substitution once means that each label in its corresponding substitution graph are shifted along the edges that leave its actual vertex.

Definition 3 (Isomorphism): Two argument vectors $X = (x_1, \ldots, x_m)$ and $Y = (y_1, \ldots, y_m)$ are *isomorphic* if

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

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

$$\begin{aligned} d &= \text{length of the cycle of } G \\ g &= \text{maximum depth of all trees} \\ &\quad\ \text{attached to the cycle of } G \end{aligned}$$

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

1. Trivially each vertex of the cycle holds the same label every d-th iteration of the mapping.
2. As the roots of all trees are involved in this cycle all trees are fed by labels of the cycle vertices.
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. 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[14]. 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. □

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

$$\begin{aligned} d &= \text{depth of the tree } + 1 \\ g &= \text{depth of the tree} \end{aligned}$$

then the following holds:

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

[14]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 already established from the beginning.

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 $k, l \geq g$.

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. $\Box$

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 = the least multiple of d_1 that is larger than d_2

b.) If G_1 does not contain a cycle and G_2 does then let
d = the least multiple of d_2 that is larger than d_1

c.) If G_1 and G_2 both contain cycles then let
d = the least common multiple of d_1 and d_2

d.) If G_1 and G_2 are both 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

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

so

$$S^{k+jd}(X) = (S_1^{k+jd}(X_1),\ S_2^{k+jd}(X_2))$$

- 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 $S_1^{k+jd}(X_1)$ 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 $S_2^{k+jd}(X_2)$ 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.

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 $S_1^{k+jd}(X_1)$ 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 $S_2^{k+jd}(X_2)$ 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:
By lemma 2 $S_1^k(X_1)$ and $S_1^{k+jd}(X_1)$ are isomorphic for $k \geq \max(g_1, g_2) \geq g_1$ because $d \geq d_1$.
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.

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

Inductive Step: $c = c_0 + 1$,
i.e. G_1 has c_0 connected components
$\Rightarrow$ because of $(*)$ the theorem holds for G_1 $\quad (**)$

a.) G_1 contains a cycle, 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 is a multiple of 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.

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. □

References

[AC89] F. Afrati and Cosmadakis. Expressiveness of restricted recursive queries. In *Proc. 21st Annual ACM Symp. on Theory of Computing*, pages 113–126, New York, 1989. ACM.

[ADJ90] R. Agrawal, S. Dar, and H.V. Jagadish. Direct Transitive Closure Algorithms: Design and Performance Evaluation. *ACM Transactions on Database Systems*, pages 427–458, September 1990.

[Agr87] R. Agrawal. Alpha: An Extension of Relational Algebra to Express a Class of Recursive Queries. In *Proceedings IEEE 3rd International Conference Data Engineering, Los Angeles, California*, February 1987.

[AJ88] R. Agrawal and H.V. Jagadish. Multiprocessor Transitive Closure Algorithms. In *Proceedings of the International Symposium on Databases*, pages 56–66, 1988.

[Akl89] S. Akl. *The Design and Analysis of Parallel Algorithms.* Prentice Hall, 1989.

[AU79] A.V. Aho and J.D. Ullman. Universality of data retrieval languages. In *Proc. 6th ACM Symp. on Principles of Programming Languages*, pages 110–120, New York, 1979. ACM.

[BR86] F. Bancilhon and R. Ramakrishnan. An Amateur's Introduction to Recursive Query Processing Strategies. In *Proceedings ACM SIGMOD 1986 Conference on Management of Data*, pages 16–52, May 1986.

[CCH90] F. Cacace, S. Ceri, and M.A.W. Houtsma. An Overview of Parallel Strategies for Transitive Closure on Algebraic Machines. In P. America, editor, *Parallel Database Systems: PRISMA Workshop, Proceedings*, pages 44–62, Noordwijk, The Netherlands, September 1990. Springer. LNCS 503.

[CW89] S.R. Cohen and O. Wolfson. Why a Single Parallelization Strategy is not Enough in Knowledge Bases. In *PODS 89*, pages 200–216, 1989.

[DR94] S. Dar and R. Ramakrishnan. A Performance Study of Transitive Closure Algorithms. In *ACM SIGMOD*, pages 454–465, 1994.

[IW91] Y.E. Ioannidis and E. Wong. Towards an Algebraic Theory of Recursion. *Journal of the ACM*, 38(2):329–381, April 1991.

[JAN87] H.V. Jagadish, R. Agrawal, and L. Ness. A Study of Transitive Closure As a Recursion Mechanism. In *Proceedings ACM SIGMOD 1987 Conference on Management of Data*, pages 331–344, 1987.

[KM91] M. Kitsuregawa and K. Matsumoto. Massively Parallel Relational Database Processing on the Connection Machine CM-2. In *Annual Report of Kitsuregawa Lab.* University of Tokyo, 1991.

[Knu73] D.E. Knuth. *The Art of Computer Programming – Sorting and Searching*, volume 3. Addison-Wesley, 1973.

[KO90] M. Kitsuregawa and Y. Ogawa. Bucket Spreading Parallel Hash: A New, Robust, Parallel Hash Join Method for Data Skew in the Super Database Computer (SDC). In D. McLeoad, R. Sacks-Davis, and H. Schek, editors, *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 210–221. Morgan Kaufmann, August 1990.

[LN89] R.J. Lipton and J.F. Naughton. Estimating the Size of Generalized Transitive Closures. In *Proceedings of the 15th International Conference on Very Large Data Bases*, pages 165–171, August 1989.

[Min93] E.M. Minty. An Algorithm for a Data Parallel Hash Join on the Connection Machine. Technical Note EPCC-TN93-02, Edinburgh Parallel Computing Centre (EPCC), March 1993.

[Nau87] J.F. Naughton. One-Sided Recursions. In *Proc. 6th ACM Symp. on Principles of Database Systems*, pages 340–348, 1987.

[NRSU89] J.F. Naughton, R. Ramakrishnan, Y. Sagiv, and J.D. Ullman. Efficient Evaluation of Right-, Left-, and Multi-linear rules. *SIGMOD Record*, 18(2):235–242, June 1989.

[Sag88] Y. Sagiv. Optimizing Datalog Programs. In J. Minker, editor, *Deductive Databases and Logic Programming.* Morgan-Kaufman, 1988.

[Ull88] J.D. Ullman. *Database and Knowledge-Base Systems*, volume 1. Computer Science Press, 1988.

[Ull89] J.D. Ullman. *Database and Knowledge-Base Systems*, volume 2. Computer Science Press, 1989.

[VB86] P. Valduriez and H. Boral. Evaluation of Recursive Queries Using Join Indices. In *Proceedings of the 1st International Conference on Expert Database Systems*, pages 197–208, April 1986.

[VK88a] P. Valduriez and S. Khoshafian. Parallel Evaluation of the Transitive Closure of a Database Relation. *International Journal of Parallel Programming*, 17(1):19–42, 1988.

[VK88b] P. Valduriez and S. Khoshafian. Transitive Closure of Transitively Closed Relations. In *Proceedings of the 2nd International Conference on Expert Database Systems*, pages 377–400, April 1988.

[WD91] C.B. Walton and A.G. Dale. Data Skew and the Scalability of Parallel Joins. In *Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing*, pages 44–51. IEEE Comput. Soc. Press, December 1991.

[WDJ91] C.B. Walton, A.G. Dale, and R.M. Jenevein. A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins. In G. M. Lohman, A. Sernadas, and R. Camps, editors, *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 537–548. Morgan Kaufman San Mateo, September 1991.

[WDY93] J.L. Wolf, D.M. Dias, and P.S. Yu. A Parallel Sort Merge Join Algorithm for Managing Data Skew. *IEEE Transactions on Parallel and Distributed Processing*, 4(1):70–86, January 1993.

[YKLH92] C. Youn, H.-J. Kim, Henschen L.J., and J. Han. Classification and Compilation of Linear Recursive Queries in Deductive Databases. *IEEE Transactions on Knowledge and Data Engineering*, pages 52–67, February 1992.

[ZT93] Th. Zurek and P. Thanisch. Processing Linear Recursive Database Queries on the Connection Machine. Technical Report EPCC-TR93-05, Edinburgh Parallel Computing Centre (EPCC), 1993.

[ZZO93] X. Zhou, Y. Zhang, and M.E. Orlowska. A Parallel Transitive Closure Algorithm for SIMD Meshes. *Australian Computer Science Communication*, 15(1):143–151, 1993.