Distributed Transitive Closure Computations: The Disconnection Set Approach*

Maurice A.W. Houtsma† Peter M.G. Apers‡ Stefano Ceri§

Abstract

This paper deals with one of the most common and important types of recursion: transitive closure. Since many real world problems reduce to generalized transitive closure computations, efficient computation is essential. To gain a significant speedup in processing, we consider distributed (i.e. parallel) computation.

By fragmenting the data beforehand according to rules stemming from the application domain, queries can be split into several independent subqueries. These subqueries are computed in parallel on only a part of the data and are more specialized in the sense that extra selections are applied on each fragment. The *disconnection set approach* introduced in this paper takes benefit from such a fragmentation; it is applicable to several queries that are based on transitive closure, such as connectivity, shortest path, and bill of materials. Moreover, it may be generalized to work for other application domains. Since we consider real world problems to deal with a large updatable volume of data, we take an algebraic approach to computation of queries. Our proposal is such that updates will, in general, not affect the fragmentation. This is also explained in the paper.

Some preliminary simulations are included in the paper as well. They show that our approach leads to a speedup that is almost proportional to the number of processors, without significant overhead.

*Partial support from NFI, a Dutch research fund, and from the LOGIDATA+ project of C.N.R Italy

†Department of Applied Mathematics, University of Twente, P.O. Box 217, 7500 AE Enschede, the Netherlands

‡Computer Science Department, University of Twente

§Dipartimento di Matematica,Universita' di Modena

Proceedings of the 16th VLDB Conference
Brisbane, Australia 1990

1 Introduction

The idea behind the use of distributed systems is that data are partitioned and allocated to several sites. The computation can then be done on a number of sites in parallel. This is possible both in the context of distributed database systems, such as [7], [27], [28], and in the context of parallel database machines, such as PRISMA [6], [21]. The main goal is, of course, to speed up processing by the use of more resources (processors). Recently, transitive closure queries have become more and more important: a body of research has been performed [1, 3, 19, 20, 24, 26], and transitive closure is being supported by some database systems [10]. The transitive closure of a relation R is defined as $\bigcup_{i=1}^{n} R^i$ (which is equal to $R \cup \pi(R \bowtie R) \cup \pi(R \bowtie \pi(R \bowtie R)) \ldots$).

Over the past decade we see a trend towards distributed computation of queries. First selections were distributed to fragments of a relation, then fragmentation was used to compute joins in a distributed way (see e.g. [9]). The next step in this process is to compute transitive closure queries in a distributed way. Since computation of the transitive closure of a relation is such a well-defined problem, studying it closer may lead to considerable insight in the use and possible benefit of parallelism.

1.1 Transitive closure in a centralized environment

Transitive closure has been studied in the context of centralized databases, where several methods have been developed in order to support efficient evaluation. In [2] it is shown how to summarize information on transitive closure and store it together with the data of the base relation; it is also possible to maintain this additional information when the base relation is updated. Similarly, [3] shows how to store additional information with the base table of a relation in order to speed up the computation of the transitive closure, and [23] shows how to materialize the transitive closure, so that queries are efficiently answered to the price of supporting a large amount of information.

It is worth noticing that e.g. [2, 24] solve only a particular type of query (that we call 'connectivity' query), while [23] may also support shortest path and bill-of-material queries. Further, it should be noticed that all these approaches apply to the problem of computing the transitive closure on a single processor; as such, they are orthogonal to the issues discussed in this paper, and they may indeed be applied on each processor independently in order to speed up its own computation.

1.2 Transitive closure in a distributed environment

Only recently, transitive closure is being studied in a distributed environment. Some research focuses on parallel evaluation in a logic programming context [17, 29]. Other research focusjes on parallel computation of matrix-based algorithms for the transitive closure [4]. We will now shortly discuss two algebraic approaches to distributed computation.

It has been noticed that the expression for the transitive closure ($\bigcup_{i=1}^{n} R^i$), may be rewritten in several ways [18]. A first straightforward approach to parallelism that uses this idea is *power splitting*. It may lead e.g. to a computation where we start with R on one processor, R^2 on an other processor, and continuously join the result on each processor with R^2. This leads to the computation of $R, R^3, R^5, \ldots$ on one processor, and $R^2, R^4, R^6, \ldots$ on the other processor. There are, however, several drawbacks to this approach. First, there is a considerable amount of data redundancy; the relation R^2 is present at two processors. Second, since there are in general several paths between a pair of nodes, paths may be generated more than once, and used more than once in the subsequent computation. This leads to many redundant computations, which is a huge disadvantage. For example, given that there exist paths of both odd and even length between a pair of nodes a and b, both processors will store the information that there exists a path between a and b, and both processors will use this information in the subsequent computations; thereby generating unnecessary (duplicate) tuples. Third, to get the final result available at one location (and to remove duplicate tuples) a union of the results has to be computed. This leads to a considerable amount of data transmission, which is usually expensive. And fourth, the transitive closure computation as sketched uses a naive approach on each processor, which is, in general, not a very efficient approach. An advantage of the approach as sketched is that it can easily be generalized for n nodes, although this enlarges the amount of redundant computation. If one tries to use a more efficient central algorithm for the computation on each node, the use of parallelism is not so straightforward any more. It results in a lot of inter-processor communication, which increases the network load and slows down computations due to the necessary synchronization between processors.

A more intricate algorithm for parallel computation of the transitive closure of a relation is described in [25]. Here, several processors are used to compute the transitive closure of a relation R. Some processors serve to store a union of particular powers R^i, while other processors serve to filter out duplicate tuples. This approach suffers from several of the drawbacks just mentioned. The removal of duplicate tuples causes a lot of inter-processor communication. Also, the computation strategy necessitates the transmission of certain powers R^i between processors; this leads to serious delays because processors have to wait for the results of other processors.

If additional information is known about the structure of the relation one wants to compute the transitive closure of, this can be used advantageously. For instance, if the relation represents a tree it may be fragmented in such a way that the transitive closure of complete subtrees is computed in parallel. The construction of the final result then requires a union of the subresults and a small number of joins. Such an approach may thus achieve a speedup that is more or less linear in the number of processors. Of course, a relation that represents a tree is a special case, which limits the applicability of the method just sketched.

From the approaches sketched above, we may learn that the advantageous use of parallel processing is not at all obvious. For a parallel process to work efficiently (in a database environment), the subtasks handed out to the processors should be relatively independent. The task to be performed on each processor should be large enough to be worthwhile shipping to this processor. The computation of duplicate tuples on different processors should be avoided. And the amount of communication between the processors should be minimal to avoid inter-process synchronization. In this paper a strategy is developed that conforms to these characteristics, by assuming a particular fragmentation of the data. The knowledge about the fragmentation of the data enables an efficient, distributed computation of transitive closure queries.

1.3 Organization of the paper

The organization of the paper is as follows. In Sec. 2 we introduce the disconnection set approach, this includes

a formal model and computation of transitive closure queries on relations consisting of n fragments. In Sec. 3 we discuss algebraic query formulation of transitive closure queries. In Sec. 4 we discuss the effect of updates on our approach. In Sec. 5 results of preliminary simulations concerning our approach are presented. Finally, Sec. 6 presents some conclusions and future research.

2 Disconnection Set Approach

A strategy that is particularly interesting when studying parallel computation of transitive closure queries, is what we call the *disconnection set approach* [13], [14]. The basic idea that underlies our approach can perhaps best be illustrated by an example. Consider a railway network connecting cities in Europe, and a question about a connection between Amsterdam and Milan. This question can be split into several parts: find a path from Amsterdam to the eastern Dutch border, find a path from the Dutch border to the southern German border, find a path from the German border to the Italian border, and find a path from the Italian border to Milan. These questions all have the same structure, but apply to only a part of the data and can be executed in parallel. Moreover, the points where one can cross a border are relatively few. This leads to a highly selective search process in an intermediate fragment, from one border city to another. Such border regions between countries are good candidates for splitting the original graph into subgraphs. In addition, some minimal, 'complementary information' about the identity of border cities and the properties of their connections has to be stored. This will be described in more detail later on.

The idea as sketched above leads us to consider a fragmentation of a graph that enables such a search process. (Remember that a relation may be viewed as the representation of a graph, with the tuples representing edges.) The graph G is partitioned into several subgraphs G_i, with each subgraph stored at a separate site. The node intersection of these subgraphs, called disconnection set, is small compared to the number of nodes in the subgraphs. For each disconnection set some 'complementary information' is stored. This information enables a reformulation of the transitive closure query into several subqueries, such that each subquery requires only one fragment. Hence, these subqueries can be processed in parallel, which leads to a considerable improvement in response time. Moreover, every subquery starts from a disconnection set and ends in a disconnection set; and as such they all consist of a selection on a query. This means, for instance, that a query consisting of a selection on a start node is reformulated in several subqueries on smaller fragments, where each subquery consists again of a selection on a node.

The structure of this section is as follows. In Sec. 2.1 a formal model for a graph is introduced, and functions that allow query formulation on this graph. In Sec. 2.2 partitioning a graph in two fragments is considered, and the reformulation of a transitive closure query into several subqueries is discussed. And in Sec. 2.3 this is generalized to partitioning a graph in n fragments.

2.1 Formal Model

In this section an abstract data structure for a graph is introduced, and functions that allow formulation of transitive closure queries on this graph. Given a set of vertices V and a set of arcs A, a directed graph G is defined as follows:

$$G = (V, A), \quad V(G) \text{ is a finite set}, \quad A(G) \subseteq V \times V.$$

Furthermore, a weight function W_G is defined that assigns a (positive) weight to arcs in G.

$$W_G : A(G) \rightarrow \mathbb{N}.$$

Now that this graph has been defined, some important functions can be defined that operate upon it. They allow the formulation of transitive closure queries. First the definition of the function *closure*:

$$\begin{aligned} closure(G) \;=\; & \{(v_1, v_2) \mid (v_1, v_2) \in A \\ & \vee \exists w \in V : ((v_1, w) \in A \\ & \wedge (w, v_2) \in closure(G))\}. \end{aligned}$$

The function *closure* results in all direct and indirect connections that exist in a graph G. The next definition is that of the function *fclosure*:

$$\begin{aligned} fclosure(G, f) \;=\; & \{\!|(v_1, v_2, c) \mid ((v_1, v_2) \in A \\ & \wedge c = W_G(v_1, v_2)) \\ & \vee (\exists w \in V : (v_1, w) \in A \\ & \wedge (w, v_2, x) \in fclosure(G, f) \\ & \wedge c = f(W_G(v_1, w), x))\}\!|. \end{aligned}$$

The function *fclosure* has as parameters a graph G and a function f. It results in a new graph which consists of the old graph G and some additional labeled arcs. These arcs represent paths in the transitive closure of the graph G, and their label is determined by using the function f on the labels of the arcs this path was created from. Usually, the label of an arc represents a weight and the function f is an arithmetic function, for example, an addition of the labels of the consisting arcs. Note that the result of *fclosure* is a multiset, which is indicated by the special brackets that are used; this means that duplicates are not removed. A last definition is that of the function *gen_closure*:

$$gen\_closure(G, f_1, f_2) \;=\; f_1(fclosure(G, f_2)).$$

The function f_2 is a function that is used in the call to *fclosure*, and is of the required type. The function f_1 is a function that operates on a multiset of labeled arcs, as is returned by *fclosure*, and returns a set of labeled arcs with some arithmetic operation performed on the labels of arcs that connect the same nodes. Note that, in general, the functions *fclosure* and *gen_closure* do not result in a finite structure.

With the help of the previously defined functions it is possible to pose all sorts of transitive closure queries. For instance, a query about the existence of a *connection* between two nodes can be answered by computing:

$$closure(G).$$

This results in the computation of the complete transitive closure of G. In the same way, a query about the *bill of material* problem (over an acyclic graph representing parts and their components) can be answered by computing:

$$gen\_closure(G, \sum, \times).$$

The application of the multiplication function results in arcs denoting a part, its subpart, and the number of times this subpart is used for the production of the part. The summation then adds the number of parts for all tuples with the same part and subpart component, to result in an arc denoting the total number of times a particular part is used for the production of the other.

Finally, a query about the *minimum cost* can be answered by computing:

$$gen\_closure(G, \min, +).$$

Here, the application of the addition function results in arcs denoting two nodes that are connected and the cost associated with this connection. By searching for the path with minimum cost for each pair of nodes, the result consists of arcs that indicate for every pair of nodes the minimum cost to go from one node to the other.

Notice that these queries are defined on an abstract graph structure. They do not imply an implementation of the actual computation. Implementation of transitive closure queries in terms of Relational Algebra is described in Sec 3.

2.2 Binary Fragmentation

Now that a formal model of a graph G and some transitive closure queries has been introduced, let us suppose that the graph G is partitioned into two fragments: G_1 and G_2. (Partitioning in n fragments is discussed in the next section.) This partitioning is done according to the following definitions:

$$G = (V, A) \qquad G_1 = (V_1, A_1) \qquad G_2 = (V_2, A_2)$$
$$A_1 \cap A_2 = \emptyset \qquad A_1 \cup A_2 = A$$
$$V_1 \cup V_2 = V \qquad V_1 \cap V_2 \neq \emptyset$$
$$DS = V_1 \cap V_2.$$

The structure DS is a so-called *disconnection set*; removal of the nodes in DS from G leaves G disconnected. Note that since the fragments have no edges in common, an edge between nodes that are part of the disconnection set DS may reside in precisely one fragment, where the choice of the fragment is arbitrary.

An example of a binary fragmentation is given in Fig. 1. It shall be clear that any possible path between nodes that reside in different fragments has to

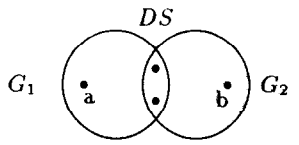

Figure 1: Binary fragmentation

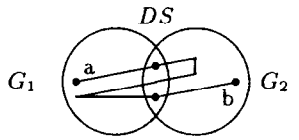

Figure 2: A zigzag path

go through the disconnection set at least once. The problem, therefore, is to find a node i in the disconnection set such that there is a connection from a given start node a via i to a given end node b. (From now on, the terms connected and connection are used both for direct and indirect connections.) However, there may be several nodes from the disconnection set on the path from a to b; a path originating from a can cross the border several times before actually staying in the fragment that includes b. Imagine, for instance, a river to form the border of two countries; a train may then cross the river several times, alternatingly using connections from each of the two countries, before reaching a main station that is connected with the bulk of the destination country. This is illustrated in Fig. 2.

To overcome problems in the case of zigzagging paths, we will use some '*complementary information*'. This 'complementary information' contains a small amount of information to solve the various transitive closure queries. It is different for different queries. For connectivity queries it contains for each node in the disconnection set the other nodes in the same disconnection set it is connected with. With the help of this 'complementary information' a query such as "Is node a connected with node b in graph G?" may now be reformulated in the following way: "Are there any nodes i and j in the disconnection set such that a is connected with i in G_1, i and j are connected in G (this is the 'complementary information'), and j is connected with b in G_2?" Of course, every node is connected with itself, hence, i may be the same node as j. Notice that the subqueries for the fragments can completely be computed in *parallel*.

To process a shortest path query, the procedure is much the same. The 'complementary information' now consists of the shortest path in G—i.e. the cost of the minimum cost path—for each pair of nodes from the disconnection set. Again, this information is small compared to the fragments. The query "What is the length of the shortest path from a to b in G?" now can be reformulated in the following way: "Find nodes i and j in the disconnection set, such that the cost of the path from a to i in G_1, plus the cost of the path from i to j in G, plus the cost of the path from j to b in G_2 is

minimal." Because every path from node a in G_1 to node b in G_2 has to go through the disconnection set, the path that is found in this way has to be the shortest path. Again, the subqueries for the fragments can be computed completely in *parallel*. Queries concerning the bill of material problem are dealt with in the same way, and described in [14].

2.3 N-ary Fragmentation

The binary fragmentation of a graph as considered in the previous section, and the solution methods for transitive closure problems, can now be generalized to a graph that consists of n fragments. Consider the following definitions:

$$G = (V, A) \quad G_1 = (V_1, A_1) \quad \ldots \quad G_n = (V_n, A_n)$$
$$\forall i, j \leq n, i \neq j : A_i \cap A_j = \emptyset$$
$$A_1 \cup A_2 \cup \ldots \cup A_{n-1} \cup A_n = A$$
$$V_1 \cup V_2 \cup \ldots \cup V_{n-1} \cup V_n = V$$
$$DS_{ij} = V_i \cap V_j,\ i \neq j$$

In the graph defined above, every arc is part of exactly one fragment. The nodes of fragments can overlap; these node intersections of the fragments are the disconnection sets. As may be seen, these definitions are a straightforward extension of the definitions for a binary fragmented graph as presented in Sec. 2.2. For the disconnection set approach to be profitable in the case of n fragments, three requirements are now introduced. They are referred to as the *disconnection set requirements*:

1. The disconnection sets should be small compared to the fragments.
2. The amount of 'complementary information' should be small.
3. Disconnection sets should be disjoint.

The rationale for these requirements is as follows:

1. The main idea underlying the disconnection set approach, as stated before in Sec. 2, is to enable a highly selective search process in each fragment separately. Therefore, the size of the disconnection set (which contains the start and end nodes of the search) should be small compared to that of the fragment: $\forall i, j$: $DS_{ij} \ll V_i$.
2. Since the disconnection fragment is to be used in the computation, it should be as small as possible.
3. To avoid replication of 'complementary information', disconnection sets should not overlap: $DS_{ij} \cap DS_{kl} \neq \emptyset \Rightarrow i = k \wedge j = l$. Moreover, if the disconnection sets were overlapping, information would also have to be stored about possible connections for nodes residing in different disconnection sets.

Before discussing query reformulation over an n-ary fragmentation, let us first introduce a concept to represent such a fragmentation.

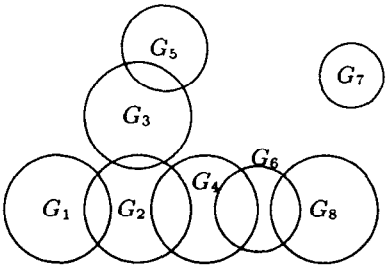

Figure 3: A fragmented relation

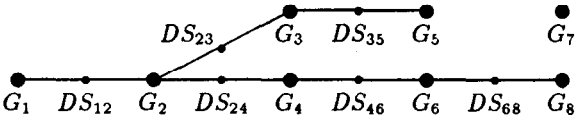

Figure 4: Fragmentation graph of fragmented relation

The fragmentation of a graph may be described by an undirected *fragmentation graph* F_G, defined as follows.

$F_G = (W, E)$ where $W = \{x \mid x$ is a disconnection set or x is a fragment$\}$

$E = \{(x, y) \mid x$ is a fragment and y is one of its disconnection sets$\}$

Hence, in the fragmentation graph F_G there exists a node for every fragment and a node for every disconnection set. There is an edge between nodes n_1 and n_2 iff n_1 is a fragment and n_2 is its disconnection set; note that the fragmentation graph is undirected. An example of a fragmentation and its fragmentation graph is shown in Figs. 3 and 4.

We initially assume that the fragmentation graph is acyclic. Let us now consider the computation of a shortest path query for the fragmentation graph as depicted in Fig. 4. If the shortest path in the original graph from node a in G_1 to node b in G_8 has to be found, the path between G_1 and G_8 in the fragmentation graph F_G has to be determined. This path P is $G_1, DS_{12}, G_2, DS_{24}, G_4, DS_{46}, G_6, DS_{68}, G_8$; where DS_{ij} stands for the disconnection set between G_i and G_j. If the shortest path from a to b in G stays within fragments on path P, it may be computed local to the fragments on this path. This means computing the shortest path from a to all nodes in the disconnection set DS_{12}, combining it with the 'complementary information' $SPDS_{12}$, computing the shortest path from all nodes in DS_{12} to all nodes in the disconnection set DS_{24}, and so on; finally taking the minimum over the costs of the paths between a and b. Note that by using the small amount of 'complementary information', the above computation computes the shortest path between a and b even if the shortest path between a and b goes back and forth between the fragments on path P (see Sec. 2.2).

There is, however, no reason why the shortest path from a to b in G should not use arcs residing in fragments G_3 and G_5. But if it does, the path is guaranteed to contain nodes that reside in the disconnection set DS_{23}. Since the shortest path between each pair of nodes from a disconnection set has already been precomputed over the *complete* graph G—and stored as 'complementary information' $SPDS_{ij}$—it suffices to compute the cost of the shortest path between nodes of DS_{12} and DS_{24} in the union of G_2 and $SPDS_{23}$.

Let us now consider a general fragmentation graph, with cycles. In this case, we add one additional constraint: shortest paths between two nodes from different disconnection sets should only use edges included in the fragment itself or edges connecting two nodes of the same disconnection set, that are therefore represented by "complementary information" of that fragment. Note that this is a natural constraint, since a path from one border to another is likely to stay in the country itself, except for small deviations at the borders.

In general, the solution strategy for finding the shortest path between an arbitrary pair of nodes a and b is as follows. One should start by looking at the fragmentation graph, locating the fragments the nodes are in, and finding all acyclic paths connecting the two fragments. Let $F_1, \ldots, F_n$ be the fragments along such a path, with $n > 1$, $a \in F_1$, $b \in F_n$. Each intermediate fragment F_i with $i > 1$ and $i < n$ (if existing), has to be traversed from the disconnection set between F_{i-1} and F_i to the disconnection set between F_i and F_{i+1}. The computation to find the paths from all nodes in the former disconnection set to all nodes in the latter disconnection set is then done over the union of the intermediate fragment and *all* its associated 'complementary information'. (With $n = 1$, a and b fall within the same fragment and the computation can be done completely local to the fragment and its 'complementary information'.) The solution is the minimum over all found paths.

What has just been shown, is that a shortest path query may be computed over an acyclic path in F_G between the fragments that contain the start node and end node of the requested path in G. This is done by taking for each fragment the union with precomputed information for all its disconnection sets. Thereby, the computation can be done in parallel for each fragment and the solution is given by a combination of the computed answers.

The process as just described has to be done for each acyclic path in the fragmentation graph that connects F_1 and F_n (when a fragment is part of several paths, the computation is, of course, only done once for that fragment). This leads to a desired property of the fragmented relations that are subject to the disconnection set approach: *their fragmentation graph should preferably be acyclic.* The fewer paths from F_1 to F_n the fragmentation graph contains, the more efficient the disconnection set approach is. In the remainder of the paper we will only consider fragmentations that have

```
function connection(From : {V}, Graph : G, To : {V}):
        V × V
begin
    New ← Graph ⋉ From;
                1=1
    Result ← New;
    while New ≠ ∅ do
        New ← π1,4 (New ⋈2=1∧1≠2 Graph)
                − Result;
        Result ← Result ∪ New
    end;
    return Result ⋉ To
                  2=1
end
```

Figure 5: Transitive closure algorithm

an acyclic fragmentation graph; these represent special cases, but are fully general in the sense that the solution process in the case of a cyclic fragmentation graph can be expressed by a number of computations for acyclic fragmentation graphs, as sketched above.

The reformulation of queries as sketched here is sound and complete. In [12], [14], and [15] we have proven this for connection, bill of material, and shortest path queries.

3 Algebraic Query Formulation

In this section algebraic query formulation over fragments is discussed. We concentrate on the connection problem, algorithms for bill of material and shortest path computations are described in [13], [14]. A graph, and, therefore, also a fragment, is assumed to be represented as a binary relation. The first attribute denotes the *from* node, the second one denotes the *to* node. This representation models a directed graph, but because an undirected graph can be represented as a directed one this is no restriction.

3.1 Connection Problem

The connection problem is essentially a simple transitive closure over a graph, where the cost of the connections is irrelevant. Therefore, we may use any kind of algorithm that is suited to the database system we are using [5]. An example of a relational program to compute the transitive closure of a relation is shown in Fig. 5. It uses a semi-naive approach with an integrated selection on the start node. The result of this program is a relation denoting which nodes from a certain set are connected with nodes from another set. Note that this program even works for cyclic relations, because of the join condition that describes that a connecting path should not return to its start node.

Now that a relational program for the computation of the connection between nodes in an arbitrary graph has been formulated, this program can be used to compute the connection between nodes in different fragments. And because these fragments are residing at different

processors, the computation can mainly be done in parallel.

Consider a graph consisting of 3 fragments: G_1, G_2, and G_3; where the fragmentation graph is acyclic. In the design process the disconnection sets DS_{12} and DS_{23} have been determined and the corresponding disconnection fragments CDS_{ij} have been computed and stored in a relation. These *disconnection fragments* represent the 'complementary information'; they are represented as binary tables, indicating all pairs of nodes in DS_{ij} that are connected in G. With $a \in G_1$ and $b \in G_3$, the existence of a path between a and b can be computed as follows:

$$\begin{aligned} C_1 &\leftarrow \text{connection}(\{a\}, G_1, DS_{12}) \\ C_2 &\leftarrow \text{connection}(DS_{12}, G_2, DS_{23}) \\ C_3 &\leftarrow \text{connection}(DS_{23}, G_3, \{b\}). \end{aligned}$$

The computation of C_1, C_2, and C_3 can be done in parallel. C_1 now contains all connections between a and nodes in DS_{12}. Similarly, C_2 contains all connections between nodes in DS_{12} and nodes in DS_{23}. Note that a join between C_1 and C_2 does not suffice to compute all connections between a and nodes in DS_{23}. As discussed in Sec. 2.2, a path can go back and forth between fragments; therefore, we have to compute a join with the precomputed disconnection fragments to find all connections. The solution is given by the following expression:

$$\pi_{1,4}(\pi_{1,4}(\pi_{1,4}(C_1 \underset{2=1}{\bowtie} CDS_{12}) \underset{2=1}{\bowtie} \pi_{1,4}(C_2 \underset{2=1}{\bowtie} CDS_{23})) \underset{2=1}{\bowtie} C_3).$$

Again, most of these joins can be computed in parallel; only the final join has to be computed on a single processor.

To simplify the expression when n fragments are considered, a slight adaptation is made. The function *connection* is not computed over the fragment itself, but over the union of the fragment with *all* its disconnection fragments; as discussed in Sec. 2.3. (Remember that the disconnection fragments are very small compared to the data fragments.) A connection between nodes in different fragments is now computed by the following expressions, which are computed in parallel:

$$\begin{aligned} C_1 &\leftarrow \text{connection}(\{a\}, G_1 \cup \bigcup_i CDS_{1i},\ DS_{12}) \\ &\vdots \\ C_n &\leftarrow \text{connection}(DS_{(n-1)n}, G_n \cup \bigcup_i CDS_{ni}, \{b\}). \end{aligned}$$

Where G_1 is the fragment a belongs to, G_n is the fragment b belongs to, and CDS_{ij} are the disconnection fragments for the disconnection sets that have a non-empty intersection with fragment G_i. The solution is given by the following expression:

$$\pi_{1,4}(\pi_{1,4}(C_1 \underset{2=1}{\bowtie} C^{\bowtie n-2}_{2-1}) \underset{2=1}{\bowtie} C_n),$$

where

$$\begin{aligned} C^{\bowtie 1}_{2-1} &= C_2 \\ C^{\bowtie i}_{2-1} &= \pi_{1,4}(C^{\bowtie i-1}_{2-1} \underset{2=1}{\bowtie} C_{i+1}). \end{aligned}$$

and $n > 2$ (with n the number of fragments). This way of formulating the solution allows us to compute $C_1, \ldots, C_n$ on n processors in parallel.

4 Updating Fragments

In this section, we show how insertion and deletion of tuples within fragments can be managed efficiently. This is required, since we assume a large updatable data collection. We consider the elementary actions *insert(a, b, w)* and *delete(a, b, w)*, where a, b are nodes and w is the weight (length) associated to the directed edge (a, b) from a to b. The *fragment update problem* is formulated as follows: given an insertion (deletion) of a tuple representing an edge in G, determine the fragment F_i where the tuple has to be inserted (deleted), and preserve the correctness of the 'complementary information' stored with F_i.

4.1 Insertion

Let us consider the elementary action *insert(a, b, w)*. We should initially determine whether it is *localized* in one fragment. This happens if there exists a fragment F_i such that $a \in V_i$ and $b \in V_i$; we assume that most updates have such a property.

With non-localized updates, we have the following situation: $a \in F_i, b \in F_j, i \neq j$ In this case, some of the present properties of fragmentation are affected (e.g. the number of adjacent fragments or the size of disconnection sets). This may either be accepted, or lead to restarting the design process. The worst case occurs if F_i and F_j are not even adjacent; in that event, we have to distinguish two situations:

- If connecting F_i to F_j introduces a cycle in the *Fragmentation Graph*, then a restart of the design process should be considered.
- If connecting F_i to F_j preserves the acyclicity in the *Fragmentation Graph*, then we can accept the insertion by adding the tuple (a, b, w) to either F_i or F_j; if the former case occurs, then $DS_{ij} = \{b\}$. No 'complementary information'is required, since DS_{ij} is a singleton set.

If we assume F_i and F_j to be adjacent, then again we have two cases:

- If neither $a \in DS_{ij}$, nor $b \in DS_{ij}$, then it is required to increase the size of DS_{ij}. If the tuple (a, b, w) is added to F_i, then b becomes part of DS_{ij}; the 'complementary information' has to be extended with information for b.
- Otherwise, let assume $a \in DS_{ij}$; then the tuple (a, b, w) must be added to F_j, and the insertion is localized in F_j (see later). Similarly, if $b \in DS_{ij}$, then the insertion is localized in F_i. If both $a \in DS_{ij}$ and $b \in DS_{ij}$, then the insertion can be localized in either fragment (the choice is arbitrary).

Finally, we consider *localized insertions*. This is the most common case in practical problems, as we assume

that fragments correspond to well-defined geographic regions with well-defined boundaries. Let $a, b \in V_i$; we consider the problem of recomputing CDS_{ij} and $SPDS_{ij}$ for one specific j such that F_j is adjacent to F_i.

Connection problem. We consider the edges of E_i, and evaluate whether a is connected to b prior to the insertion. If so, then the operation has no effect on CDS_{ij}. Otherwise, let R_a be the set of nodes of DS_{ij} that can be reached from a, R_b the set of nodes of DS_{ij} that reach b. Then, we compute the new value CDS'_{ij} from the old value CDS_{ij} as:

$$CDS'_{ij} = CDS_{ij} \cup (R_a \times R_b)$$

Shortest path problem. We consider the edges E_i and evaluate whether a is connected to b through a path shorter than w prior to the insertion. If so, then the operation has no effect on $SPDS_{ij}$. Otherwise, it is required to compute the shortest paths connecting nodes of DS_{ij} and substitute the new values if they improve over the old values (hence, (a, b) is part of the new path). This is easily achieved by computing the shortest paths between b and nodes of DS_{ij}; then summing up w. This procedure may be shortened if some sufficient conditions occur; for instance, the computation can be immediately halted when the update occurs in an area which is "sufficiently remote" from the disconnection set, or when w is greater than the maximum shortest path in $SPDS_{ij}$, or finally halted when the partial shortest paths including (a, b) all become greater than the corresponding shortest path stored in $SPDS_{ij}$.

Both with the connection problem and the shortest path problem, some further processing is required if CDS_{ij} or $SPDS_{ij}$ are modified as effect of an update in fragment F_i. In fact, the 'complementary information' about disconnection sets describes global properties about the original graph G, and not just about F_i. Thus, inserted tuples into CDS_{ij} or $SPDS_{ij}$ have to be communicated to the computer storing fragment F_j, in order to be propagated into symmetric structures CDS_{ji} and $SPDS_{ji}$.

Further, these changes in connectivity or shortest path of nodes in the disconnection set can cause additional changes in CDS_{ij} and $SPDS_{ij}$; they are evaluated as new insertions into the 'complementary information,' through methods discussed above. If any further changes arises in CDS_{ji} or $SPDS_{ji}$, it has to be propagated back to F_i; this process is iterated until the computation reaches a fixpoint (guaranteed by the finiteness of the graph G and by monotonicity of operations involved). A a matter of fact, even other auxiliary information of disconnection sets DS_{jk}, wiht $k \neq i$ might be affected by a change in CDS_{ji} or $SPDS_{ji}$. However, especially in the case of an acyclic fragmentation graph such a propagation is quite unlikely.

4.2 Deletion

Let us consider the elementary action *delete(a, b, w)*. Obviously, deletions are all *localized*; let $a \in N_i, b \in N_i$; we consider the problem of recomputing CDS_{ij} and $SPDS_{ij}$.

Connection problem. We consider the remaining edges E_i, and evaluate whether a is still connected to b. If so, then the operation has no effect on CDS_{ij}. Otherwise, it is required to recompute CDS_{ij}, as it is not easy to understand otherwise the implications of the deletion operation.

Shortest path problem. We consider the remaining edges E_i, and evaluate whether a is connected to b through a path shorter than w. If so, then the operation has no effect on $SPDS_{ij}$. Otherwise, it is required to recompute $SPDS_{ij}$, as it is not easy to understand otherwise the implications of the deletion operation. Once again, this might be unnecessary if sufficient conditions occur; for instance, the computation can be immediately halted when the update occurs in an area which is "sufficiently remote" from the disconnection set.

We envision applications in which queries are much more frequent then updates, all update operations are localized into one fragment, and changes to 'complementary information' occur very rarely; in these cases, the test for existence of side effects of insertions and deletions on auxiliary information can be performed in polynomial time for all considered update operations, and is largely dependent on the size of the disconnection set.

5 Simulation Results

The ideas presented in the previous sections regarding the use of disconnection sets for the computation of transitive closure queries seem promising; and to get a better idea of the benefits it seems worthwhile to investigate them in a more practical way, by conducting simulations. This has been done in the context of the PRISMA database machine [21]. This is a distributed main-memory database system running on a multi-processor system. Such a system seems especially suited to profit from the proposed strategy. Since the processors are connected by a high speed network, parallel processing may be used profitably. The disconnection set approach is, of course, suited for standard distributed databases as well, but the benefits may be seen more clearly in the context of a tightly coupled distributed database system.

For the simulations we used relations that were randomly generated as explained shortly. We have not restricted our simulations to trees and DAGs (as e.g. in [4]) since these present special cases; instead, we focussed on graphs in general. In this paper we will focus on the results of the simulations and not discuss the simulation model and implementation of relational operations (this can be found in [22], [12]). Just let us note that the PRISMA machine is based on a message-passing paradigm (no shared memory), where each processor has 16 Mbyte of private memory and the processors are connected by a high speed network. The implementation of the relational operations is hash-based,

	R50-1	R100-1	R200-1	R25-2	R50-2	R100-2
DSS_1	2.97	10.93	43.10	13.86	54.17	159.37
PS_2	2.30	7.81	40.29	28.48	103.39	423.90
RS_7	2.80	10.47	56.77	54.69	230.09	—

	R50-1	R100-1	R200-1	R25-2	R50-2	R100-2
R^T	147	433	1717	409	1730	5455

Table 1: Simulation results for actual relations

since this proves to be fastest in a main-memory environment [11].

The structure of this section is as follows. In Sec. 5.1 a comparison is made between central computation and parallel computation; this section explains some of the problems parallel approaches have to cope with. And in Sec. 5.2 the simulation results for the disconnection set approach are presented, and the benefits of this approach are made obvious.

5.1 Central versus Parallel

For the execution of the simulations, binary relations were generated in a random way. This was done according to two parameters: the connection average (i.e. the chance that there is a direct connection between two nodes), and the number of tuples in the start relation. To have an honest comparison, it was assumed that every simulation started with the relation R on a single node; transmitting this relation to other nodes is part of the computation strategy and the costs are taken into account.

In Table 1 some typical results for a fast single-processor algorithm and two parallel strategies are presented. The single-processor algorithm is the so-called delta smart squaring algorithm [8], [12]; it combines semi-naive and logarithmic approaches. The parallel strategies are a strategy based on power splitting (PS), as described in Sec. 1.2, and an optimized version of an algorithm based on [25], which uses 7 processors for the computation of the transitive closure. In Table 1 the subscript of the algorithms indicates the number of processors used, the name of the relations is derived from the number of tuples and the connection average of the relations. Hence, RS_7 means that the results for the RS-algorithm are shown, where 7 processors are used by the algorithm; and R100-2 means that the relation R that was used consisted of 100 tuples, with a connection average of 2. The numbers in Fig. 1 indicate the response time for each algorithm on each relation, the cardinality of the transitive closure of the relations is indicated as well.

The results as presented in Fig. 1 might seem surprising: the benefit of parallel processing seems to be virtually nil. Especially when the graph is not sparsely connected the behaviour of the parallel algorithms deteriorates rapidly. To gain better insight in the factors that determine the efficiency of parallel computation and explain the disappointing results of the parallel algorithms, the processor activities have been put into a

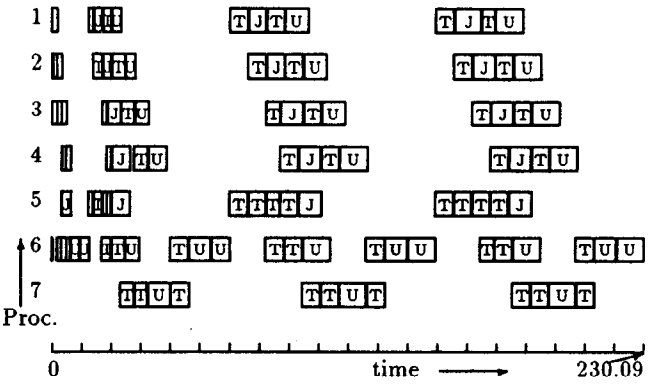

Figure 6: Processor activity RS_7-algorithm for relation 50-2

graphical form. Fig. 6 presents the activity for the RS_7-algorithm. In this diagram, the relational operations on each processor are shown over a period of time. The operations are marked with a character indicating their purpose: U for union, J for join, and T for transmit.

From Fig. 6 we may conclude that in the RS_7-algorithm the processors are used only a part of the time. Much time is spent on waiting for results of other processors. The time spent on waiting for transmission leads to a low amount of operational parallelism. Only for short intervals of time the processors are all working in parallel. Another drawback of this approach is the production of duplicate tuples; this results in a computation that goes on longer than necessary. The processor activity for the PS-algorithm is not shown here; this algorithm does achieve a reasonable amount of parallelism, but at the cost of generating many redundant tuples and thereby a longer computation time.

All in all, the comparison between central and parallel processing does not seem hopeful for the effective use of parallelism in the computation of transitive closure queries. It seems that one can achieve a high degree of operational parallelism—by making the computations on the different processors more or less independent—at the cost of generating many duplicate (i.e. useless) tuples. Or one can use many processors that have to exchange information and, thereby, achieve only a low degree of operational parallelism due to necessary synchronization between processors. In the next section simulations for the disconnection set approach are presented, to see if this approach does a better job in effectively using parallel processing.

5.2 Disconnection Set Approach

Since the disconnection set approach avoids the generation of duplicate tuples as much as possible, it is beneficiary in a central environment as well. Therefore, we compare a parallel implementation of the disconnection set approach with a single processor one. Now,

	R_3-178	R_5-309	R_8-16141
Single processor	6.59	16.73	1940.62
Multi processor	3.29	5.09	336.91

Figure 7: Simulation results disconnection set approach

the results may be compared without giving one of the approaches some unforeseen advantage. The transitive closure computation on each processor is done using a semi-naive approach with integrated selection, as depicted in Fig. 5.

In Fig. 7 the results of the simulations are shown. The approach has been tested for relations consisting of 3, 5, and 8 fragments (as indicated by the subscripts in the figure). For the relations that consist of 3 or 5 fragments the selection assumed two start nodes and two end nodes. For the relation consisting of 8 fragments the selection assumed four start nodes and four end nodes; the disconnection sets contain 2 to 6 nodes. Since relations with a high connection average proved to be the most difficult for parallel approaches, this type of relations is used for the simulations; the value of the connection average used in the simulations is 2.

The names for the relations are chosen to represent the number of fragments and the total number of tuples in the relation. Hence, R_5-309 indicates a relation consisting of 5 fragments and 309 tuples. The cardinality of the final result of the transitive closure query is 4 for the relations consisting of 3 or 5 fragments, and 16 for the relation consisting of 8 fragments. Of course, in the multi-processor algorithms as much processors were used as there were fragments.

From the results as depicted in Fig. 7, we may conclude that the disconnection set approach indeed is very promising. For instance, the use of 8 processors results in a computation that requires only 18% percent of the time required for a central computation. For the relation that consists of 8 fragments, the processor activity is depicted in Fig. 8. It shows that the degree of operational parallelism is very high indeed. For a large interval of time all processors are working in parallel. At the end of the computation, the joins of the results per fragment with the disconnection fragments are executed. These joins require only a few processors and a relatively short period of time.

The benefit of the disconnection set approach, as was made clear by the simulations, is due to a number of aspects. First, the computations on the processors are almost completely independent. A processor can keep on working right until it finishes its task, only then it might have to wait before being able to transmit the results. Second, no tuple is generated on more than one processor. This avoids a great number of redundant computations—an important difference with other parallel approaches. And third, the computation does not depend on the diameter of the graph (the length of the longest path) as do the central and parallel approaches

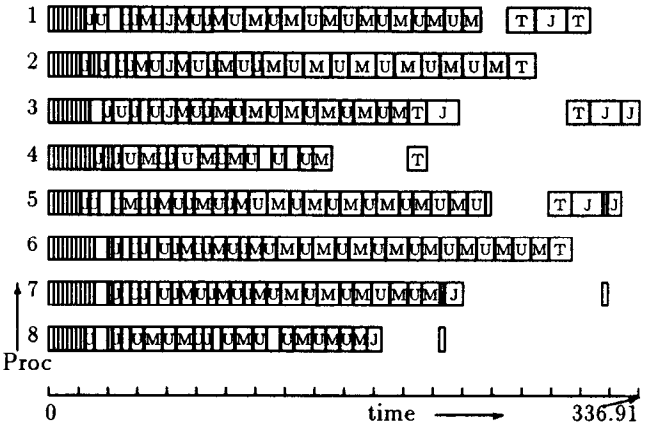

Figure 8: Processor activity for R_8-16141

sketched before. It now depends on the diameter of the fragments, which is, therefore, in the design process best chosen as directly related to the number of fragments the graph is partitioned in. A possibility that has not been investigated, but seems worthwhile, is to choose different transitive closure algorithms for each fragment, depending on its connection average.

6 Conclusions and Future Research

In this paper it was shown that the profitable use of parallel processing in the computation of transitive closure queries is a difficult problem. More often than not parallel strategies fail to achieve a reasonable gain in performance. For a parallel strategy to be really worthwhile, it should strive for equally-sized independent tasks; this means that inter-processor communication should be avoided. It is also very important to minimize the number of redundant tuples generated. The development of this kind of beneficiary parallel strategies requires knowledge about the structure of the relation and about the type of queries.

The disconnection set approach conforms to these relevant characteristics. It allows a reformulation of transitive closure queries into similar queries that require only one fragment. This reformulation has been proven correct (in [12, 14, 15]) and the small amount of 'complementary information' needed for this reformulation has been discussed. The disconnection set approach works for queries based on transitive closure, such as connectivity, shortest path, and bill of material (this paper concentrated on connectivity). Algebraic query formulation for these queries has been described, to show that they can actually be implemented on a relational database system. We have also described how updates may influence the 'complementary information' and how they can be handled efficiently. An interesting aspect of the disconnection set approach is that its benefits may also be used in a central environment, although no use can be made then of parallel processing.

We have not discussed fragmentation design in this paper. Here, the focus was on efficiently computing transitive closure queries in parallel. At the moment we are studying fragmentation design, where we try to concentrate on fragmentations that have an acyclic fragmentation graph. As noted in the paper, the disconnection set is fully general and applicable to all sorts of fragmentations, but it is much more efficient when the fragmentation graph is acyclic. In [13] we have described a combinatorial algorithm for such a fragmentation design, but we feel that in general knowledge of the application domain is required for a good fragmentation design. Note that many application domains conform to the characteristics that are desirable when using the disconnection set approach, for instance, public transportation, part-subpart.

The simulations in the paper showed that a number of methods described before did not gain a significant speed up by parallel processing in our multiprocessor environment. Simulations concerning the disconnection set approach showed that it did achieve a significant speedup; due to the division of work among several processors, and the use of extra selections on each subquery (stemming from the disconnection sets on the path from start node to end node). In fact, this last aspect is also important in a central environment and will lead to a performance improvement even in such a central environment [22]. We will conduct more simulations investigating these issues, and plan to make a full comparison with other approaches.

References

[1] AGRAWAL, R. 'Alpha: an extension of relational algebra to express a class of recursive queries,' in *IEEE Transactions on Software Engineering*, Vol. 14, No. 7, July 1988, pp. 879–885.

[2] AGRAWAL, R., BORGIDA, A. AND JAGADISH, H.V. 'Efficient management of transitive relationships in large data and knowledge bases,' in *Proc. Int. Conf. on Management of Data, ACM-SIGMOD*, Portland, Oregon, 1989, pp. 253–262.

[3] AGRAWAL, R. AND JAGADISH, H.V. 'Efficient search in very large databases,' in *Proc. 14th Int. Conf. on Very Large Databases*, Los Angeles, 1988, pp. 407–418.

[4] AGRAWAL, R. AND JAGADISH, H.V. "Multiprocessor transitive closure algorithms," in *Proc. Int. Symp. on Databases in Parallel and Distributed Systems*, Austin, Texas, Dec. 5–7 1988, pp. 56–66.

[5] APERS, P.M.G., HOUTSMA, M.A.W. AND BRANDSE, F. "Processing recursive queries in relational algebra," in *Data and Knowledge (DS-2), Proc. of the 2nd IFIP 2.6 Working Conf. on Database Semantics, Albufeira, Portugal, Nov. 3–7, 1986*, R.A. Meersman and A.Cy. Sernadas (eds.), North Holland, 1988, pp. 17–39.

[6] APERS, P.M.G., KERSTEN, M.L., AND OERLEMANS, H. "PRISMA database machine: A distributed main-memory approach," in *Advances in Database Technology-EDBT'88*, J.W. Schmidt, S. Ceri and M. Missikoff (eds.), Lecture Notes in Computer Science **#303**, Springer-Verlag, 1988, pp. 590–593.

[7] BERNSTEIN, P.A., GOODMAN, N., WONG, E., REEVE, C.L., AND ROTHNIE, J.B. "Query processing in a system for distributed databases (SDD-1)," *ACM Transactions on Database Systems 4* 6, 1981, pp. 602–625.

[8] BREHLER, J. "Transitive closure operation in a relational database," M.Sc. Thesis, University of Twente, the Netherlands, March 1988.

[9] CERI, S. AND PELAGATTI, G. *Distributed Databases, principles & systems*, McGraw-Hill, 1985.

[10] CERI, S., GOTTLOB, G. AND TANCA, L. *Logic Programming and Databases*, Springer-Verlag, 1990.

[11] DEWITT, D.J., KATZ, R.H., OLKEN, F., SHAPIRO, L.D., STONEBRAKER, M.R., AND WOOD, D. "Implementation techniques for main memory database systems.," in *Proc. Int. Conf. on Management of Data, ACM-SIGMOD*, Boston, USA, June 18–21, 1984, pp. 1–8.

[12] HOUTSMA, M.A.W. *Data and Knowledge Base Management Systems: Data Model and Query Processing*, Ph.D. Thesis, University of Twente, Enschede, the Netherlands, Nov. 1989.

[13] HOUTSMA, M.A.W., APERS, P.M.G., AND CERI, S. "Parallel computation of transitive closure queries on fragmented databases," Technical Report INF-88-56, University of Twente, the Netherlands, Dec. 1988.

[14] HOUTSMA, M.A.W., APERS, P.M.G., AND CERI, S. "Distributed transitive closure computations: the disconnection set approach," Technical report INF-89-12, University of Twente, the Netherlands, Oct. 1989.

[15] HOUTSMA, M.A.W., APERS, P.M.G., AND CERI, S. "Complex Transitive closure queries on a fragmented graph," Submitted for publication.

[16] HOUTSMA, M.A.W., VAN KUIJK, H.J.A., FLOKSTRA, J., APERS, P.M.G. AND KERSTEN, M.L. "A logic query language and its algebraic optimization for a multiprocessor database machine," University of Twente, the Netherlands, Technical report INF-88-52, Dec. 1988.

[17] HULIN, G. "Parallel processing of recursive queries in distributed architectures" in *Proc. 15th Int. Conf. on Very Large Databases*, Amsterdam, 1989, pp. 87–96.

[18] IOANNIDIS, Y.E. "On the computation of the transitive closure of relational operators," *Proc. of the 12th Int. Conf. on Very Large Data Bases*, Kyoto, Japan, Aug. 1986, pp. 403–411.

[19] IOANNIDIS, Y. AND RAMAKRISHNAN, R. 'Efficient transitive closure algorithms,' in *Proc. 14th Int. Conf. on Very Large Databases*, Los Angeles, 1988, pp. 382–394.

[20] JAGADISH, H.V., AGRAWAL, R. AND NESS, L. 'A study of transitive closure as a recursion mechanism,' in *Proc. Int. Conf. on Management of Data, ACM-SIGMOD*, 1987. pp. 331–344.

[21] KERSTEN, M.L., APERS, P.M.G., HOUTSMA, M.A.W., VAN KUIJK, H.J.A., AND VAN DE WEG, R.L.W. "A distributed, main-memory database machine," in *Proc. of the 5th Int. Workshop on Database Machines*, Karuizawa, Japan, Oct. 5–8, 1987; and in *Database Machines and Knowledge Base Machines*, M. Kitsuregawa and H. Tanaka (eds.), Kluwer Academic Publishers, 1988, pp. 353–369.

[22] KLEINHUIS, G. AND OSKAM, K.R. "Evaluation and simulation of parallel algorithms for the transitive closure operation," M.Sc. Thesis, University of Twente, the Netherlands, May 1989.

[23] LARSON, P-A. AND DESHPANDE, V. 'A file structure supporting traversal recursion,' in *Proc. Int. Conf. on Management of Data, ACM-SIGMOD*, Portland, Oregon, 1989, pp. 243–252.

[24] LU, H. 'New strategies for computing the transitive closure of a database relation,' in *Proc. 13th Int. Conf. on Very Large Databases*, Brighton, 1987, pp. 267–274.

[25] RASCHID, L. AND SU, S.Y.W. "A parallel strategy for evaluating recursive queries," in *Proc. of the 12th Int. Conf. on Very Large Data Bases*, Kyoto, Japan, Aug. 1986.

[26] ROSENTHAL, A., HEILER, S., DAYAL, U. AND MANOLA, F. 'Traversal recursion: a practical approach to supporting recursive applications in *Proc. Int. Conf. on Management of Data, ACM-SIGMOD*, 1986, pp. 166–176.

[27] STONEBRAKER, M. AND NEUHOLD, E. "A distributed version of Ingres," *Proc. 2nd Berkeley Workshop Distributed Data Management and Computer Networks*, May 1977, pp. 19–36.

[28] TANDEM DATABASE GROUP "NonStop SQL, a distributed, high-performance, high-availability implementation of SQL," Tandem report, April 1977.

[29] WOLFSON, O. "Sharing the Load of Logic-program Evaluation," in *Proc. Int. Symp. on Databases in Parallel and Distributed Systems*, Austin, Texas, Dec. 5–7 1988, pp. 46–55.