Complex Transitive Closure Queries on a Fragmented Graph*

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

Abstract

In this paper we study the reformulation of transitive closure queries on a fragmented graph. We split a query into several subqueries, each requiring only a fragment of the graph. We prove this reformulation to be correct for shortest path and bill of material queries. Here we describe the reformulation for an abstract graph, elsewhere we have described an actual implementation of our approach and some promising simulation results.

We view the study of distributed computation of transitive closure queries as a result of the trend towards distributed computation. First selections were distributed to fragments of a relation, then fragmentation was used to compute joins in a distributed way, and now we are studying distributed computation of transitive closure queries. This should result in a deeper insight into the use and possible benefit of parallelism. Our work may be used in ordinary distributed databases as well as advanced multiprocessor database machines, such as PRISMA.

Although this research was started to efficiently use distributed computation, it turns out to be beneficiary in a central environment as well. This is due to the introduction of extra selections, stemming from an appropriate fragmentation. This leads to extra focus on relevant data.

1 Introduction

Over the past decade there has been a constant development and proliferation of so-called deductive databases, and systems that couple logic programming and relational databases. This was caused by the wish to have an easier way of expressing complex problems, enable easy modification of programs for e.g. what-if analysis, and the need for more expressive power in the query language (see e.g. [26]). The major research topic arising from the forementioned development has been the study of optimization strategies for recursive queries (e.g. [4, 10, 11, 15]).

Since any realistic application will still be dealing with a large amount of data, the use of full-fledged relational databases systems as underlying server is indispensable. This has influenced research on recursive query optimization by focusing on bottom-up optimization strategies, such as magic sets [6]. Main point of such optimization strategies is to push selections into the evaluation as much as possible. Another, more recent, important research direction is the use of distributed processing (parallelism) to speed up computation-intensive recursive queries [9, 8, 14, 18, 27] Since queries are becoming more and more complex, an effective optimization strategy needs some knowledge about the application domain. This is especially so for optimization in

*This research was partially supported by NFI, a Dutch research fund, and the Logidata+ project of the National Research Council of Italy

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

‡Computer Science Department

§Dipartimento di Matematica, Universita' di Modena, Italy

distributed (parallel) systems. In this paper we assume such particular knowledge, in the form of a predetermined fragmentation. Our strategy should thus be viewed as one out of a range of strategies; where the query optimizer chooses a particular strategy based on knowledge about the application domain or heuristic rules.

In a logic programming context parallelism may be achieved by duplication of the program onto several processors, where each program deals with only part of the data. This is the approach followed e.g. in [30], where rules of the program are extended with evaluable predicates. The output of such an evaluable predicate is used to determine the other processors data is send to. Although there is no need for synchronization (a possible source for delay), load balancing schemes may only be found for special classes of programs [13]. Another approach is described in [20]; it uses a top-down approach and relies heavily on inter-processor communication, thus requiring a lot of synchronization.

In a relational database context the use of distributed processing focuses on a special operation, viz. transitive closure. (In fact, this is not such a limitation, since logic programming usually focuses on forms of linear recursive programs, which essentially depict transitive closure problems.) In [3] parallel versions of matrix-based transitive closure algorithms are studied. They essentially study the simple connection problem (is node a connected to node b?), but their approach is unfeasible for the more complicated bill of material and shortest path problems studied in this paper. Another approach to parallel computation of transitive closure is described in [28]. However, this approach leads to serious synchronization problems and much inter-processor communication that negatively influences the performance [15, 17].

Since our concern is with large databases, distribution of data during query processing is prohibitively expensive. Even in a closely coupled distributed database, such as PRISMA [23], with a fast network connecting the processors, distributing a large amount of data at run-time is very expensive and will form a major bottleneck. Therefore, data has to be distributed *a priori*, presumably with the help of some knowledge about the application domain.

In this paper we consider the relations in a database to depict graphs, and investigate the kind of fragmentation of such graphs that would lead to a favourable distributed computation of transitive closure queries (since the transitive closure of a relation is, in general, much too large to be of interest, we focus on queries concerning a selection on start node and end node). Given such a fragmentation we study the reformulation and computation of complex transitive closure queries. We present some theorems about this reformulation and prove them to be correct. The characteristics of our approach, called the *disconnection set approach*, are that it requires no inter-processor synchronization, almost no inter-processor communication, is applicable to several transitive closure queries (we focus on shortest path and bill of material), and leads to a highly selective search process on each fragment. This means that our approach makes very efficient use of distributed processing, since all major bottlenecks are avoided. Actually, our approach turns out to be beneficiary in a central environment as well [15, 24]. The price we have to pay for all this is a small amount of precomputed information, and the need for some semantic knowledge concerning the application domain. As already stated, we feel that this last requirement is unavoidable for any approach that is to make real benefit from distributed computation.

We should note that our approach is completely orthogonal to research on central computation of transitive closure queries (e.g. [1, 2, 21, 22, 25]). We do not presuppose specific

implementations of transitive closure algorithms on each processor, instead we may use any favourable central algorithm. In this way we may combine the best of both worlds.

We view the study of distributed computation of transitive closure queries as a result of the trend towards distributed computation. First selections were distributed to fragments of a relation, then fragmentation was used to compute joins in a distributed way [12], and now we are studying distributed computation of transitive closure queries. This should result in a deeper insight into the use and possible benefit of parallelism. Our work may be used in ordinary distributed databases [29] as well as advanced multiprocessor database machines, such as PRISMA [23, 5].

The structure of the paper is as follows. In Sec. 2 we introduce the disconnection set approach, study reformulation of transitive closure queries on fragmented graphs and prove it correct. In Sec. 3 we present conclusions and discuss future research.

2 Disconnection Set Approach

In this section we introduce a strategy for studying parallel computation of transitive closure queries, called the *disconnection set approach* [15, 18]. 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 the shortest path between Amsterdam and Milan.[1] This question may 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 may be executed in parallel. We assume that the data are naturally fragmented by state, for instance, in a Dutch, German, Austrian, and Italian database that can be accessed through a distributed database system. Moreover, we assume that the points where one can cross a border are relatively few.

This fragmentation leads to a highly selective search process in an intermediate fragment, consisting of determining the properties of connections from the origin to the first border, then between borders of two subsequent intermediate fragments, and finally from the last border to the destination. These questions all have the same structure, but apply to only one fragment of the data and may be executed in parallel. 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 may be processed in parallel, which leads to a considerable improvement in response time.

[1] Note that in pure Datalog this question cannot be expressed, however, several extensions of Datalog exist that allow such a query [7].

The structure of this section is as follows. In Sec. 2.1 a formal model for a graph and functions that allow query formulation are introduced. In Sec. 2.2 reformulation of shortest path and bill of material queries into several subqueries is discussed for a graph partitioned in two fragments. In Sec. 2.3 this is generalized for a graph partitioned 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.

$$W_G : A(G) \rightarrow I\!N.$$

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*:

$$closure(G) = \{(x, y) \mid (x, y) \in A \quad \vee \quad \exists z \in V : ((x, z) \in A \wedge (z, y) \in closure(G))\}.$$

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*:

$$\begin{aligned} fclosure(G, f) \quad = \quad & \{\!|(x, y, c) \mid ((x, y) \in A \wedge c = W_G(x, y)) \\ & \vee \quad (\exists z \in V : (x, z) \in A \wedge (z, y, c') \in fclosure(G, f) \\ & \wedge \quad c = f(W_G(x, z), c'))\}\!|. \end{aligned}$$

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. A last definition is that of the function *gen_closure*:

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

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, thus resulting 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.

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 this 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:

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

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 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

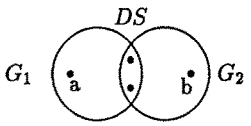

Figure 1: Binary fragmentation

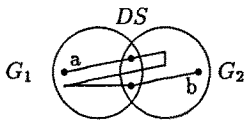

Figure 2: A zigzag path

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 with these zigzagging paths, we will use some '*complementary information*'. This 'complementary information' contains a small amount of information to solve the various transitive closure queries, which is different for each type of query. In the next section we study the reformulation of shortest path and bill of material queries on a binary fragmentation with the help of 'complementary information.'

2.2.1 Shortest Path

For a shortest path query, the 'complementary information' 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. Since the disconnection sets are small, this information is small compared to the fragments. The query "What is the length of the shortest path from a to b in G?" may now 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. Note that the subqueries for the fragments can be computed completely in *parallel.*

If we assume that nodes a and b are both in graph G_1, Theorem 1 can be formulated. It states that the shortest path is the path with minimum associated weight from all paths that exist, either direct, or via the disconnection set. The cost of the shortest path between a and b in graph G is denoted by $sp_G(a, b)$.

Theorem 1 *For a and $b \in V_1$ the following holds:* $sp_G(a,b) = \min(sp_{G_1}(a, b), \min_{i,j \in DS}(sp_{G_1}(a, i) + sp_G(i, j) + sp_{G_1}(j, b)))$.

Proof: For every path from a to b there are precisely two possibilities: it consist solely of arcs from G_1, or it contains at least one arc from G_2.

For paths of the first category we may state that the cost of the shortest path is $sp_{G_1}(a,b)$.

Paths of the second category have a structure $a, x_1, \ldots, x_n, b$. Choose the first node p_1 on this path such that $p_1 \in DS$. Similarly, choose the last node $p_2 \in DS$ on this path. The cost of $path(a,b)$ is given by $cost_{G_1}(a,p_1) + cost_G(p_1,p_2) + cost_{G_1}(p_2,b)$, and hence the cost of the shortest path is $sp_{G_1}(a,p_1) + sp_G(p_1,p_2) + sp_{G_1}(p_2,b)$.

Therefore, it follows that the cost of the shortest path from a to b in G (with a and b both in G_1) is $\min(sp_{G_1}(a,b), \min_{i,j \in DS}(sp_{G_1}(a,i) + sp_G(i,j) + sp_{G_1}(j,b)))$. ■

Now assume that a and b are nodes in different fragments, without loss of generality we may assume that $a \in V_1$ and $b \in V_2$. Theorem 2 states that the cost of the shortest path from a to b is comprised of the cost of the path from a to a node i in the disconnection set, plus the cost of the path from i to a node j in the disconnection set that is connected with b, plus the cost of taking the path from j to b.

Theorem 2 *For $a \in V_1$ and $b \in V_2$ the following holds:* $sp_G(a,b) = \min_{i,j \in DS}\{sp_{G_1}(a,i) + sp_G(i,j) + sp_{G_2}(j,b)\}$.

Proof: All paths from a to b have the structure $a, x_1, \ldots, x_n, b$, and $\exists\, x_i \in DS$. Choose the first node p_1 on the minimal cost path mcp such that $p_1 \in DS$. In the same way choose the last node $p_2 \in DS$ on this path. The costs of mcp are given by $cost_{G_1}(a,p_1) + cost_G(p_1,p_2) + cost_{G_2}(p_2,b)$. And since mcp is minimal, this is equivalent to $sp_{G_1}(a,p_1) + sp_G(p_1,p_2) + sp_{G_2}(p_2,b)$. Hence, the cost of the minimal path is equivalent to $\min_{i,j \in DS}\{sp_{G_1}(a,i) + sp_G(i,j) + sp_{G_2}(j,b)\}$ ■

2.2.2 Bill of Material

To process a bill of material query, we have to be more careful with the reformulation of queries. First of all, bill of material queries require *all* paths between part a and b to be computed. And second, since bill of material queries require addition and multiplication we have to be cautious not to introduce redundant information. This is illustrated by Fig. 3, where the arcs and labels represent the part-subpart relationship. Hence, part i is used twice in the construction of part a, part j is used three times in the construction of part i, etc. The bold arcs (from a to i and from j to k) are part of fragment G_1, the other ones are part of fragment G_2; nodes i, j, and k are part of the disconnection set, as indicated by the ellips. As we can see, there is an arc from i to j in G_2. If we would take this arc into account when computing the 'complementary information' for nodes i and j in the disconnection set and we would use the same reformulation as before it would lead to a wrong answer. We would now compute the cost for the path from a to i in G_1, use the 'complementary information' for the connection (i,j), and the path from j to b in G_2. But we would also compute the cost for the path from a to i in G_1 and from i to b in G_2. Hence, the same information (namely the cost of the path through (i,j)) is used twice, and, therefore, we get a wrong answer.

An obvious solution would then be to store for every pair of nodes x, y from the disconnection set the number of times y is part of x in G *minus* the number of times y is part of x in G_1 or G_2. Thus, we would avoid taking the same path into account more than once. Hence, the 'complementary information,' denoted by *bomdf(x, y)*, that is stored for each pair of nodes x, y from the disconnection set would then be: $nop_G(x,y) - nop_{G_1}(x,y) - nop_{G_2}(x,y)$; where

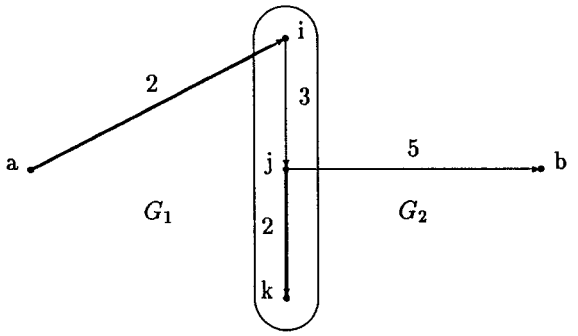

Figure 3: Bill of material example

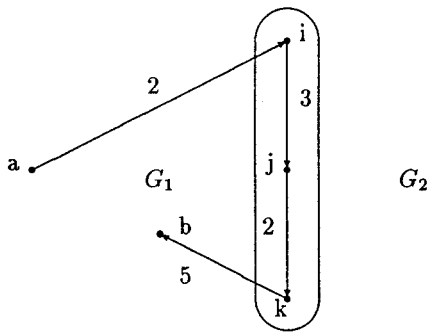

Figure 4: Modified Bill of material example

$nop_{G_i}(x, y)$ denotes the number of times part y is used in the construction of part x in fragment G_i. However, if we change the example by moving node b and arc $(j, k, 2)$ into G_1 as in Fig. 4 we may see a problem. Given the 'complementary information' computed as described above, we would *not* find the correct answer for the path from a to b; this is due to the fact that the path from i to k is completely contained in G_1.

To overcome the problems as just described, we will store three sets of 'complementary information' for each disconnection set; one computed over the complete graph minus connections that are completely in G_1 ($nop_{\neg G_1}$), one computed over the complete graph minus connections that are completely in G_2 ($nop_{\neg G_2}$), and one computed over the complete graph (nop_G). For nodes a and b both in G_1 we may now compute the bill of material problem over G_1 and use the 'complementary information' $nop_{\neg G_1}$; this solves any problems for situations as depicted in Fig. 4. For nodes a and b in different fragments we may formulate the bill of material problem as follows:

1. Find all paths in G_1 from a to a node in the disconnection set where there are no other nodes from the disconnection set on this path.

2. Find all paths in G_2 from nodes in the disconnection set to b where there are no other

nodes from the disconnection set on this path.

3. Use the 'complementary information' nop_G to link the results together, and sum the costs for all paths from a to b.

If we assume that a and b are nodes in the same fragment G_1 we have the following theorem, which states that the number of parts b in a is equal to the number of parts b in a when we consider G_1 plus the number of parts b in a when we consider the paths that have a subpath partly outside G_1.

Theorem 3 *For a and $b \in V_1$ the following holds: $nop_G(a, b) = nop_{G_1}(a, b) + \sum_{i,j \in DS} nop_{G_1}(a, i) \times nop_{\neg G_1}(i, j) \times nop_{G_1}(i, b)$, with $i \neq j$, and there are no nodes from DS on $path(a, i)$ or $path(j, b)$.*

Proof: The paths between nodes a and b can be classified in two disjunct categories: paths that are completely contained in G_1 (i.e. all edges are in G_1) and paths that are not. The first category results in the expression $nop_{G_1}(a, b)$ to compute the number of times b is part of a. Now we have to compute a similar number for the second category, i.e. all paths that have a subpath that is partly outside G_1. Choose x as the first node of DS on such a path and y as the last node of DS on the same path (and since the subpath is partly outside G_1 and the graph is acyclic $x \neq y$). The cost of such a path is computed by $nop_{G_1}(a, x) \times nop_G(x, y) \times nop_{G_1}(y, b)$. But since paths from a to b that are completely in G_1 have already been taken into account when computing the result for the first category, these have to be excluded. Therefore, the cost of the paths from x to y that are completely contained in G_1 have to be subtracted and the expression changes in $nop_{G_1}(a, x) \times nop_{\neg G_1}(x, y) \times nop_{G_1}(y, b)$. This should now be summed over all pairs of nodes from the disconnection set DS.

Since all paths from a to b fall in exactly one of the two categories we are sure to have used all paths in the computation. And since the construction is such that each path is used exactly once, we are sure it is counted only once. Hence, $nop_G(a, b) = nop_{G_1}(a, b) + \sum_{x,y \in DS} nop_{G_1}(a, x) \times nop_{\neg G_1}(x, y) \times nop_{G_1}(y, b)$, with $x \neq y$ and there are no nodes from DS on $path(a, x)$ or $path(y, b)$. ■

If we assume that $a \in V_1$ and $b \in V_2$ we have the following theorem, which states that the number of parts b in a is equal to the multiplication of the number of parts i in a in G_1, the number of parts b in j in G_2, and the number of parts j in i in G. As explained before, i should be the first node of the disconnection set on $path(a, i)$ and j should be the last node of the disconnection set on $path(j, b)$.

Theorem 4 *For $a \in V_1$ and $b \in V_2$ the following holds: $nop_G(a, b) = \sum_{i,j \in DS} nop_{G_1}(a, i) \times nop_G(i, j) \times nop_{G_2}(j, b)$, where there are no nodes from DS on $path(a, i)$ and $path(j, b)$.*

Proof: All paths from a to b have to go through the disconnection set DS. Choose x as the first node of DS on such a path and y as the last node of DS on the same path. (x and y are allowed to be the same, with an associated cost of 1.) The number of parts x in a can be computed completely in G_1: $nop_{G_1}(a, x)$. Similarly, the number of parts b in y can be computed completely in G_2: $nop_{G_2}(y, b)$. The number of parts b in a for the paths with x and y as first respectively last node in DS is now given by $nop_{G_1}(a, x) \times nop_G(x, y) \times nop_{G_2}(y, b)$. Since the constraint that x is the first node of DS on the path and y is the last node of DS on the

path divides all paths into distinct classes, we are sure that every path is used exactly once in the computation. We can thus simply sum over all nodes in the disconnection set: $nop_G(a, b) = \sum_{i,j \in DS} nop_{G_1}(a, i) \times nop_G(i, j) \times nop_{G_2}(j, b)$, where there are no nodes from DS on $path(a, i)$ and $path(j, b)$. ■

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:

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

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, is to enable a highly selective search process in each fragment separately. Therefore, the size of the disconnection set (which contains the start nodes 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 'complementary information' 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: $DS_{ij} \cap DS_{kl} \neq \varnothing \Rightarrow i = k \wedge j = l$. Moreover, if the disconnection sets were overlapping, information would also have to be stored about possible connections for nodes residing in different disconnection sets; since we could then leave a fragment through one disconnection set, and enter it again through another one.

Before discussing query reformulation over an n-ary fragmentation, let us first introduce a concept to represent such a fragmentation. The fragmentation of a graph may be described by an undirected *fragmentation graph* F_G, defined as follows.

$$F_G = (N, E) \text{ where } N = \{N_i \mid N_i \text{ is a fragment}\}$$

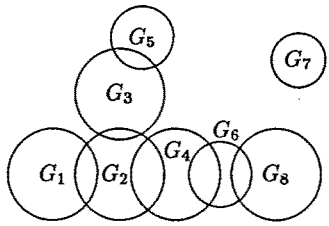

Figure 5: A fragmented relation

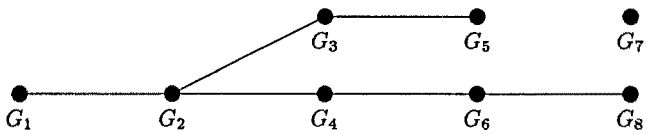

Figure 6: Fragmentation graph of fragmented relation

$$E = \{(N_i, N_j) \mid DS_{ij} \neq \varnothing\}$$

Hence, in the fragmentation graph F_G there exists a node for every fragment and there is an edge between nodes N_1 and N_2 overlap, i.e. they have a non-empty disconnection set. Note that the fragmentation graph is undirected. An example of a fragmentation and its fragmentation graph is shown in Figs. 5 and 6

Let us now consider the computation of a shortest path query for the graph as depicted in Fig. 6. (Note that the fragmentation graph of this graph is acyclic, the case of cyclic fragmentation graphs is discussed shortly.) 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, G_2, G_4, G_6, G_8. 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} over G_1 and its 'complementary information' for DS_{12}, computing the shortest path from all nodes in DS_{12} to all nodes in the disconnection set DS_{24} over G_2 and the 'complementary information for its disconnection sets, 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'—it suffices to compute the cost of the shortest path between nodes of DS_{12} and DS_{24} over the union of G_2 and the 'complementary information' for all its disconnection sets.

In general, the solution strategy for finding the shortest path between an arbitrary pair of nodes a and b is as follows. One looks at the fragmentation graph, locates the fragments the nodes are in, and finds 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 done over the union of the intermediate fragment F_i 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.

This shows 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.

Since this procedure has to be followed for each acyclic path connecting F_1 to F_n (when a fragment is part of several paths, the computation is, of course, only done once for that fragment), a desired property of the fragmented relations is that the fragmentation graph is acyclic. The fewer paths from F_1 to F_n the fragmentation graph contains, the more efficient the disconnection set approach is. Although acyclic fragmentations represent special cases, they are fully general in the sense that the solution process in the case of a cyclic fragmentation graph uses the same kind of procedure. We only add one additional constraint to deal with cyclic fragmentation graphs: shortest paths between two nodes that are part of different disconnection sets should only use edges included in the fragment itself, or edges connecting two nodes of the same disconnection set (these are therefore part of the 'complementary information' for that disconnection set). 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.

The reformulation of queries as sketched here is sound and complete. This will be proven now for the shortest path problem, presuming an acyclic fragmentation graph.

2.3.1 Shortest Path

Given an acyclic fragmentation graph, the theorems of Sec. 2.2 and their proofs can be generalized to the case of n fragments. This is described now for the shortest path query. Theorem 5 states that the shortest path from a to b, with $a \in V_1$ and $b \in V_n$, may indeed be computed as has just been sketched. (Note that $sp_G(x,x) = 0$ for any $x \in V$.) The notation DS^- is introduced for the first disconnection set on the path from G_1 to G_n in F_G, and DS^+ for the last. $G \setminus G_- \setminus G_+$ denotes the graph consisting of the original graph G minus all arcs from the fragment node a is in (G_-) and all arcs from the fragment node b is in (G_+). By G_x^* the union of fragment G_x with the precomputed information (disconnection fragments) for all its associated disconnection sets is meant.

Theorem 5 *For $a \in V_1$ and $b \in V_n$ where F_G is acyclic, the following holds:* $sp_G(a,b) = \min_{i \in DS^-,\ j \in DS^+} (sp_{G_-^*}(a,i) + sp_{G \setminus G_- \setminus G_+}(i,j) + sp_{G_+^*}(j,b))$

Proof: Suppose *mcp* is a path from a to b with minimal cost. This path *mcp* has structure $a, x_1, \ldots, x_n, b$. Choose the last node p_1 on this path such that $p_1 \in DS^-$. Similarly, choose the first node p_2, such that $p_2 \in DS^+$, The cost of the path from p_1 to p_2 is minimal, and by construction equal to $sp_{G \setminus G_- \setminus G_+}(p_1, p_2)$. For the path from a to p_1 we may notice the following. Although it may contain any arcs from the graph G, by construction of G it always has to leave and enter G_- by the same disconnection set. Hence, we may ignore arcs that are outside G_- and instead use the disconnection fragments: they precisely store the relevant information concerning paths from nodes in the disconnection set to other nodes in the same disconnection set. The cost of the path from a to p_1 is thus equal to $sp_{G^*_-}(a, p_1)$. Similarly, the cost of the path from p_2 to b is equal to $sp_{G^*_+}(p_2, b)$. Therefore, the cost of the minimal cost path from a to b is indeed given by $\min_{i \in DS^-, j \in DS^+}(sp^*_-(a, i) + sp_{G \setminus G_- \setminus G_+}(i, j) + sp_{G^*_+}(j, b))$. ■

The practical use of Theorem 5 is that it allows us to split the computation of a shortest path query into three parts. The computation for G_- and the computation for G_+ may now be done with the help of the disconnection fragments; and by recursively applying Theorem 5 on the remaining graph $(G \setminus G_- \setminus G_+)$ this computation may also be subdivided in the same way. In [16, 17] this has been illustrated by giving Relational Algebra programs for computation of transitive closure queries on an actual database system.

3 Conclusions

In this paper we have studied the reformulation of transitive closure queries on a fragmented graph. The disconnection set approach enabled us to split a query into several subqueries, each requiring only a fragment of the graph. We have proven this reformulation correct for shortest path and bill of material queries. (Note, by the way, that other approaches usually restrict themselves to simple connection queries).

In contrast to approaches to parallelization in the field of logic databases, we do not change our program to determine where computed data should reside. Instead, we assume a fragmentation beforehand and use the original program on just a part of the data. We feel that some knowledge about the application domain—here in the form of fragments that might represent countries in a railroad network—is required to really benefit from distributed processing. (In logic strategies such as [30] this knowledge might be put into the choice of a hash-function.)

We have not discussed how to obtain a good fragmentation, this is done in [19], where also a generalization of the disconnection set approach is introduced, called parallel hierarchical evaluation. This generalization is based on real-life observations, assuming a high-speed network connecting the fragments. How to deal with the effect of updates on the 'complementary information' is discussed in [18]. Implementations of transitive closure algorithms for shortest path and bill-of-material problems are described in [15, 17].

In [15, 18] we have reported on simulations that show that the disconnection set approach indeed achieves a remarkable speed-up in processing on a parallel database machine such as PRISMA. In fact, it also is beneficiary in a central environment. This can be explained by noting that whereas the original query contains a selection on the start and end node, the disconnection set approach introduces such a selection *for each fragment*. So, selections are not only pushed into the query in our approach, but extra selections are introduced as well. This leads to extra focus on relevant data.

References

[1] AGRAWAL, R., BORGIDA, A. AND JAGADISH, H.V. 'Efficient management of transitive relationships in large data and knowledge bases,' in *Proc. ACM-Sigmod Conference*, Portland, Oregon, 1989, pp. 253–262.

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

[3] 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.

[4] 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 Conference on Database Semantics, Albufeira, Portugal, Nov. 3–7, 1986*, R.A. Meersman and A.C. Sernadas (eds.), North Holland, 1988, pp. 17–39.

[5] 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.

[6] BANCILHON, F., MAIER, D., SAGIV, Y., AND ULLMAN, J.D. "Magic sets and other strange ways to implement logic programs," *Proc. ACM SIGMOD-SIGACT Symp. on Principles of Database Systems*, 1986.

[7] CACACE F., CERI S., CRESPI-REGHIZZI C., TANCA L., AND ZICARI R. "Integrating object oriented data modeling with a rule-based programming language", in *Proc. ACM-Sigmod Conference*, Atlantic City, USA, May 1990.

[8] CACACE, F., CERI, S., AND HOUTSMA, M.A.W. "A survey of parallel execution strategies for transitive closure and logic programs," Technical Report, University of Twente–Politecnico Milano, Sep. 1990. Submitted for publication.

[9] CERI, S., CACACE, F., AND HOUTSMA, M.A.W. "An overview of parallel strategies for transitive closure on algebraic machines," *Proc. Workshop on Parallel Database Systems*, Noorwijk, the Netherlands, Sept. 1990, Lecture Notes in Computer Science, Springer-Verlag.

[10] CERI, S. AND TANCA, L. "Optimization of systems of algebraic equations for evaluating Datalog queries," in *Proc. of the 13th Int. Conf. on Very Large Data Bases*, Brighton, Sept. 1987, pp. 31–42.

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

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

[13] COHEN, A.R. AND WOLFSON, O. "Why a single parallelization strategy is not enough in knowledge bases," Technical report no. 561, Technion, Israel, June 1989.

[14] GANGULY S., SILBERSCHATZ A., AND TSUR S. "A framework for the parallel processing of Datalog queries," *Proc. ACM-Sigmod Conference*, Atlantic City, USA, May 1990.

[15] 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.

[16] 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.

[17] 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, Nov. 1989.

[18] HOUTSMA, M.A.W., APERS, P.M.G., AND CERI, S. "Distributed transitive closure computations: The disconnection set approach," in *Proc. 16th Int. Conf. on Very Large Data Bases*, Brisbane, Australia, Aug. 1990.

[19] HOUTSMA, M.A.W., CACACE, F., AND CERI, S. "Parallel hierarchical evaluation of transitive closure queries," Technical Report, University of Twente, Nov. 1990.

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

[21] 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.

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

[23] 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.

[24] 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.

[25] LARSON, P-A. AND DESHPANDE, V. 'A file structure supporting traversal recursion,' in *Proc. ACM-SIGMOD*, Portland, Oregon, 1989, pp. 243–252.

[26] NAQVI, S. AND TSUR, S. *A logic language for data and knowledge bases*, CS Press, 1989.

[27] NEJDL W., CERI S., AND WIEDERHOLD G. "Evaluating recursive queries in distributed databases," Tech. Rep. 90-015, Politecnico di Milano, submitted for publication.

[28] 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.

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

[30] 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.