Data fragmentation for parallel transitive closure strategies

Maurice A.W. Houtsma* Peter M.G. Apers Gideon L.V. Schipper
University of Twente, P.O. Box 217, 7500 AE Enschede, the Netherlands

Abstract

A topic that is currently inspiring a lot of research is parallel (distributed) computation of transitive closure queries. In [10] the disconnection set approach has been introduced as an effective strategy for such a computation. It involves reformulating a transitive closure query on a relation into a number of transitive closure queries on smaller fragments; these queries can then execute independently on the fragments, without need for communication and without computing the same tuples at more than one processor.

Now that effective strategies as just mentioned have been developed, the next problem is that of developing adequate data fragmentation strategies for these approaches. This is a difficult problem, but of paramount importance to the success of these approaches.

We discuss the issues that influence data fragmentation. We present a number of algorithms, each focusing on one of the important issues. We discuss the pros and cons of the algorithms, and we give some results of applying the algorithms to different types of graphs. This last aspect shows to what respect the algorithms indeed conform to the goals we set out.

1 Introduction

The transitive closure is an important extension of the functionality of database systems. For example, in a database containing information about edges and the cost of edges between nodes, one can formulate questions like "Is A connected to B?" and "What is the cost of the shortest path between A and B?". And in a database storing information about parts, one can express bill-of-material questions. This functionality has been proposed in the context of logical query languages [6, 17] and in the context of the relational algebra [1, 13]. Independent of the context, at the implementation level one needs an algorithm to efficiently process the transitive closure; many algorithms have been proposed, e.g. [16].

Efficient evaluation of the transitive closure is still a problem. Therefore, much research is currently taking place into its parallel computation [3, 8, 12, 15, 19, 21, 22]. For a good overview of parallel strategies for computing transitive closure queries, we refer to [5].

In [10, 11] we introduced the *disconnection set approach* as a way of attacking this efficiency problem in a parallel environment. The disconnection set approach is based on the divide-and-conquer principle. The basic idea is to split the relation that represents a connection network into a number of fragments. In addition to the fragmentation, some connectivity information for nodes on the intersection of fragments is computed and stored. A connection query like, "Is A connected to B?" can now be replaced by n independent transitive closure queries on individual fragments, followed by one final query on a very small relation. The characteristics of this approach are: no communication between the transitive closure operations, high selectivity for the searches on the fragments, and no redundant computations.

Because of these characteristics, the disconnection set approach is well suited for parallel evaluation of the transitive closure. (Also in a centralized environment it performs better than other algorithms.) For good fragmentations, it gives a linear speed-up. The choice of the fragmentation is vital for the performance; some obvious requirements come to mind. First of all, it seems a good idea to have fragments of more or less the same size, so that the parallel computations take the same time. And second, the disconnection sets should be rather small to restrict the search and to limit the amount of precomputed information. These requirements may be conflicting.

This paper will address the problem of fragmenting a relation to make the (parallel) computation of the transitive closure based on the disconnection set approach efficient. To better understand this design problem we will focus on transportation networks. These are characterized by clusters of nodes with a rather high internal connectivity rate, while these clus-

*The research of Maurice Houtsma has been made possible by a fellowship of the Royal Netherlands Academy of Arts and Sciences; email: houtsma@cs.utwente.nl

ters are loosely interconnected. Three requirements that have to be fulfilled by a fragmentation are formulated and three different fragmentation strategies are presented, each emphasizing one of these requirements. Some test results are presented to show the performance of the various fragmentation strategies.

The paper is organized as follows. In Sec. 2 we shortly describe the disconnection set approach again, and then go into the characteristics of fragmentations that may affect the performance of the disconnection set approach. In Sec. 3 we discuss the type of graphs that we focus on, and then introduce three algorithms for fragmenting a graph; each algorithm focuses on a particular characteristic. In Sec. 4 we give test results of the developed algorithms on different types of graphs. Finally, in Sec. 5 we give conclusions and discuss future research.

2 Disconnection set approach and data fragmentation

In this section, we will shortly introduce the disconnection set approach [10, 11]. Then we will discuss the characteristics of the disconnection set approach that influence fragmentation design. There are several options for fragmentation strategies, depending on the issue that is considered to be most relevant. As we do not yet know which of these issues has the greatest influence on performance, we will develop several algorithms—each focusing on one of the characteristics—later on in Sec. 3

2.1 Disconnection set approach: a sketch

The disconnection set approach assumes an initial data fragmentation based on application's semantics. Consider a railway network connecting cities in Europe, and a question about the shortest connection between Amsterdam and Milan. Assume that data are naturally fragmented by country (e.g., Holland, Germany, and Italy). Also assume that the border points between countries are relatively few. The above question can then 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. All these queries have the same structure; they apply only to a fragment of the database, and can be executed in parallel. Post-processing is required to assemble the shortest path between the initial and final city, given all shortest paths produced within single fragments. The intuitive idea behind the disconnection set approach is illustrated in Fig. 1, for a query concerning the connection between a node in G_1 and a node in G_8.

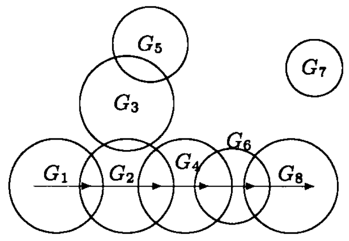

Figure 1: Intuitive idea of the disconnection set approach

We assume that the base relation R stores the connection information[1]; R is partitioned into n fragments R_i ($1 \leq i \leq n$) each stored at a different computer or processor. This fragmentation induces a partitioning of G into n subgraphs G_i. *Disconnection sets* DS_{ij} are given by $G_i \cap G_j$ (they are thus sets of nodes). We assume that the number of nodes belonging to disconnection sets is much less than the total number of nodes in G.

In order to make the above approach feasible,[2] it is required to store in addition some *complementary information* about the identity of border cities and the properties of their connections; these properties depend on the particular path problem considered. For instance, for the shortest path problem it is required to precompute the shortest path among any two cities on the border between two fragments. Complementary information about the disconnection set DS_{ij} is stored at both sites storing the fragments R_i and R_j.

An important, but not strictly necessary, property of a fragmentation is to be *loosely connected*: this corresponds to having an acyclic graph G' of components G_i. Formally, $G' =< N, E >$ has a node N_i for each fragment G_i and an edge $E_{ij} = (N_i, N_j)$ for each nonempty disconnection set DS_{ij}. In Fig. 2 the fragmentation graph for the graph shown in Fig. 1 is shown. Intuitively, if the fragmentation graph is loosely connected, then it is easier to select fragments involved in the computation of the shortest path between two nodes. In particular, for any two nodes in

[1]R represents a directed graph, where each tuple represents an edge of the graph, possibly with an associated weight

[2]I.e., to guarantee that answers are correct and precise.

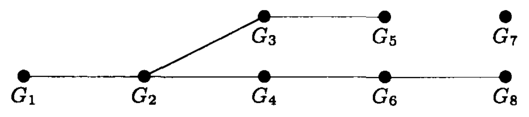

Figure 2: Fragmentation graph of fragmented relation

G there is only one chain of fragments G_i such that the first one includes the first node, the last one includes the last node, and remaining fragments in the chain connect the first fragment to the last fragment. However, for many practical problems (such as the European railway network itself) such property does not hold.

In [10] it is shown that, if the fragmentation is loosely connected, the shortest path connecting any two cities is found by involving in the computation only the computers along the chain of fragments connecting them[3]. If the fragmentation is not loosely connected, it is required to consider all possible chains of fragments independently for solving the query.[4] Obviously, if the source and destination are within the same fragment, the query can be solved by involving only the computer storing data about that fragment, including all complementary information about disconnection sets stored at that fragment. In practice, this has the nice implication that queries about the shortest path of two cities in Holland can be answered by the Dutch railway computer system alone, even if the path goes outside the Dutch border.

Along a chain of length n, query processing is performed in parallel at each computer. Each subquery determines a shortest path per fragment; note that disconnection sets introduce additional selections in the processing of the recursive query, they act as intermediate nodes that must be mandatorily traversed. The final processing combines all shortest paths obtained from the various processors with the complementary information, and selects the shortest one among them. This final processing is effectively a sequence of binary joins between a number of very small relations.

An important speed-up factor is due to the reduced number of iterations required to compute each recursive query independently. The number of iterations required before reaching a fixpoint is given by the maximum diameter of the graph; if the graph is fragmented in n fragments G_i of equal size, the diameter of each subgraph is highly reduced.

Note that neither communication nor synchronization is required during the first phase of the computation; for evaluating the recursive subquery on a fragment any suitable single-processor algorithm may be chosen; it is also possible to use some other parallel method. Only at the end of the computation, communication is required for computing the final joins. These joins will have relatively small operands (since the disconnection sets are small) and pipelining may be used for their computation.

The disadvantage of the disconnection set approach is mainly due to the pre-processing required for building the complementary information and to the careful treatment of updates. Complementary information is different for each type of path problem. As long as updates are not too frequent, the pre-processing costs may be amortized over many queries.

2.2 Important issues for fragmentation design

From the description of the disconnection set approach, we may distil three issues that are of major importance to its performance: size of the disconnection sets, size of the fragments, and the existence of cycles in the fragmentation graph.

Disconnection sets When finding a path from node x in fragment F_i to node y in fragment F_m, a number of disconnection sets DS_{ij}, DS_{jk}, etc. are encountered. These disconnection sets act as some sort of keyhole: only paths travelling through this keyhole have to be examined. Starting from this keyhole a sort of 'magic cone' is build (cf. magic sets).

As the disconnection sets function as selection criteria, the smaller they are the better. Selectivity is already important in the case of ordinary joins, but for transitive closure computations it is even more important. As transitive closure computations are very computation intensive, and each answer from one iteration is used in the next iteration, the higher the selectivity (i.e. the smaller the disconnection set) the better.

Fragments An important aspect in parallel computation is balancing the workload. If the workload is evenly spread over the processors, they can all

[3]Note that the shortest path might include nodes *outside* the chain, however, their contribution is precomputed in the complementary information.

[4]If the fragmentation graph becomes very complex and contains many routes from one fragment to another, a technique called ***Parallel Hierarchical Evaluation*** can be used to avoid problems [12]

finish at more or less the same time. Any eventual final computation that needs access to the result of several processors can then start immediately at the end.

The time needed for computing a transitive closure is determined by the number of iterations of the transitive closure algorithm and the size of the intermediate results. The number of iterations depends on the diameter of a fragment (the number of edges constituting the longest path), the size of intermediate results depends on the connectivity of the graph. Assuming that the connectivity and the diameter of the fragments are similar, the number of tuples in a fragment is a good indication for the workload of a processor.

Cycles When considering a path from node x in fragment F_i to node y in fragment F_m, the optimal situation is to find precisely one chain of fragments connecting F_i to F_m. Therefore, to minimize the number of cycles; a loosely connected fragmentation is to be preferred.

We have now discussed the three most important issues when fragmenting a relation to enable the disconnection set approach. Unfortunately, these issues are not independent; they may even conflict with each other. For instance, the graph may have such a structure that we either fragment it in a way that gives small disconnection sets but many cycles, or in a way that gives no cycles but large disconnection sets. It is not obvious which of these fragmentations should be preferred. Another example of conflicting requirements may be small disconnection sets vs. fragments of similar size. Small disconnection sets minimize computation on a fragment, fragments of similar size maximize the amount of parallelism (i.e. processors actually working at the same time). Which of these issues is most relevant is impossible to say beforehand, but will have to be deduced from actual experiments. For instance, if the underlying database system has a good support of pipelining in query processing, the issue of fragment size may become less relevant as processes do not need to wait for the final assembling of the answers.

We have now indicated which three issues are of main importance when fragmenting a relation. It should also be clear that there is no easy way of predicting which issue is most important. The issues clearly intertwine, and decisions which criterion to focus on should be based on experiments.

In the next section we will discuss a number of fragmentation strategies. Each strategy will focus on a particular issue. Once these fragmentation strategies have been developed and implemented, we can start experiments to decide on the relevance of the issues discussed.

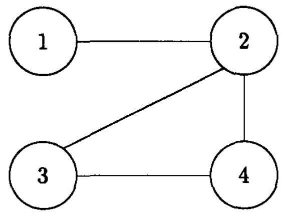

Figure 3: Transportation graph consisting of 4 clusters

3 Data fragmentation strategies

It turns out that fragmenting an arbitrary graph in a meaningful way is a highly nontrivial problem. Although a sketch of a particular graph may give us the idea that it can easily be fragmented, once we try to write an algorithm for it, it turns out to be a real problem.

We will, at first, restrict ourselves to a specific type of graph, which is particularly suited for parallel evaluation using the disconnection set approach. We call this type of graph *transportation graph*; an example of it is given in Fig. 3. It consists of a number of clusters; each cluster in itself is highly connected, but the connection between clusters is very loose. Transportation graphs occur frequently in real applications, consider e.g. local train networks per region and fast intercity trains connecting the regions, or local telephone networks (heavily connected) with a few optic fibres connecting these local networks. Because transportation graphs are very common in applications, we feel that our initial restriction to this type of graph is justified. We will, however, also test the algorithms on arbitrary graphs, later in Sec. 4.2.

For the transportation graph in Fig. 3, there are some obvious candidate nodes for the disconnection sets; the clusters should become fragments, and the nodes on the inter-cluster connections should end up in the disconnection sets. Our first idea was to come up with a simple graph-theoretical algorithm that made use of the characteristics of transportation graphs. We did this by investigating the k-connectivity of a graph (this is the smallest number of node-distinct paths between any pair of nodes from the graph). The nodes whose removal would increase the k-connectivity of the graph were marked as 'relevant' nodes, with the idea that a number of them could be selected to form disconnection sets. However, even for 'simple' graphs as depicted in Fig. 3 we would run into problems; as soon as the fragmentation graph contains cycles, the k-connectivity is influenced by paths taking detours through other fragments. Also, algorithms like this are very computation intensive, as all possible combinations of nodes and paths have to be taken into account.

We will discuss three other algorithms we developed. Each is based on a different characteristic as discussed in Sec. 2.2

3.1 Center-based algorithm

The center-based algorithm focuses on achieving a balanced workload. Ideally, all the fragments created by this algorithm should require the same amount of computation when computing the local transitive closure from the starting disconnection set to another disconnection set. As described before, the characteristics that determine the amount of computation for a fragment are diameter and number of tuples. Generally, it will not make a big difference which of these characteristics we put first when fragmenting data. However, the algorithm we present is flexible and allows us to choose either one if it does make a difference.

The algorithm starts by determining so-called 'centers' in the graph. These centers should be thought of as some kind of gravity points in the graph, very much like spiders in a web. Starting from these centers the fragments are gradually constructed (a first sketch of this approach was given in [12], and a similar idea was, for different purposes, pursued in [2]). Centers are determined based on the number of neighbour nodes (transitively),as expressed by the following formula:

$$grade(i)+a\sum_j nb(j,1)+a^2\sum_j nb(j,2)+a^3\sum_j nb(j,3)$$

with *grade(i)* the number of edges adjacent to i, $nb(j,d)$ the grade of node j at d edges from i and $a<1$. This formula is a variation on the status score [9].

The number of centers that is chosen may depend on factors such as the number of processors available, or on the application.

Once it has been decided which nodes are centers, the algorithm iterates over them and repeatedly adds edges to the fragments. In the first step, edges connecting to the center are added. In subsequent steps, edges are added to a fragment if they connect to edges that have already been assigned to that fragment.

```
/* Input: graph $G = (V, E)$; Output: fragments
  $G_1, \ldots, G_n$; $V = \bigcup_i V_i$; $E = \bigcup_i E_i$ */
$\{c_1, \ldots, c_n\}$ := determine_centers($G$);
/* Initialisation */
for $i$ from 1 to $n$ do
  $V_i := \{c_i\}$;
  $E_i := \{ (x,y) \mid x = c_i \vee y = c_i\}$
od;
$k := 1$;
$E := E \setminus \bigcup_i E_i$;
while $E \neq \emptyset$ do
  $E_k := E_k \cup \{(x,y) \mid (x,y) \in E \wedge (x \in V_k \vee y \in V_k)\}$;
  $V_k := \{x \mid (x,y) \in E_k \vee (y,x) \in E_k\}$;
  $E := E \setminus E_k$;
  $k := (k \bmod n) + 1$
od
```

Figure 4: Center-based fragmentation

The algorithm is adaptable in the sense that the iteration over the centers may be based on the resulting diameter of the fragment or on the number of tuples already included. In the first case, one addition of edges (in fact, a relational join between intermediate result and the relation modeling the graph) is done at each iteration. In the second case, the fragment with the least number of edges is chosen for expansion until another fragment becomes the smallest. The first variant of this algorithm is shown in Fig. 4.

In practice, it may occur that centers are relatively close to each other, possibly leading to large disconnection sets. If there is a notion of topology in the network (as is often the case, cf. countries in a railroad network), an optimization is possible. In this case the centers are spread over the graph, using the notion of topology.

3.2 Bond-energy algorithm

In this section we describe an algorithm that focuses on fragmenting a relation in such a way that the node intersections of fragments will be small. The algorithm is a variant of the well-known Bond-energy algorithm [7]. It uses an adjacency-matrix to denote the graph being fragmented. Columns of this matrix are reordered in such a way that nodes that are closely related are put closely together. In this way, clusters are formed along the diagonal of the matrix. By splitting the matrix in such a way that the number of 1's (representing direct connections) outside each cluster

$$\left[\begin{array}{ccc|ccc}
1 & 1 & 0 & 0 & 1 & 0 \\
1 & 1 & 1 & 0 & 1 & 0 \\
0 & 1 & 1 & 0 & 0 & 0 \\
\hline
0 & 0 & 0 & 1 & 0 & 1 \\
\underline{1} & \underline{1} & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 1 & 1 & 1
\end{array}\right]$$

Figure 5: Fragmenting a matrix

is small, the disconnection sets are kept small. Let us describe this now in more detail.

The algorithm starts with an adjacency-matrix M, where each entry $M[i,j]$ is 1 if there exists a direct connection between i and j, and 0 otherwise. Each entry $M[i,i]$ is also made 1. An arbitrary column is now chosen as the first column of the matrix, next to it the column is placed that maximizes the inner product of the two columns (defined as $\sum_{k=1}^{n} x_{ki}x_{kj}$, for columns x_i, x_j). Then, in each step a new column is chosen and put either on the left of the columns that have been placed, or on the right, or in between two columns, whichever maximizes the sum of the inner products of all the placed columns. The outcome of this procedure depends on the column that was chosen as the first one to be placed. Therefore, it has to be iterated over *all* the columns. Finally, the matrix with the greatest sum of inner products is the result.

Now that the matrix is clustered, the next step is to split it. Fragmenting the relation represented by the matrix means that we define blocks of contiguous columns (rows) as fragments. The 1's for the columns of a block that fall outside the corresponding rows for this block, are the connections with other fragments; their number indicates the size of the disconnection sets. In Fig. 5 a 6×6-matrix is shown. If nodes 1–3 are grouped together, there are 2 connections with nodes outside the block, both with node 5. If instead nodes 1–4 are grouped together, there are 3 connections with nodes outside the block, with nodes 5 and 6. For minimizing the size of the disconnection sets, the first fragmentation is to be preferred.

In general, we will want to split the relation into more than 2 fragments; in this case it is not so simple to determine the size of the disconnection sets while splitting the matrix. If we decide that the first k columns should form a fragment, we can determine the overlap of this fragment with the remaining part of the matrix. However, this overlap is only an indication of the size of the *union* of all disconnection sets for this fragment, not for the size of a disconnection set itself. This because fragmenting the rest of the matrix may lead to a number of overlaps with the first fragment, each contributing to the overlap just found.

Examining all possible ways to split a given matrix of n columns into m blocks of contiguous columns would require far too much work, as there are $\binom{n-1}{m-1}$ possibilities. We have therefore implemented a simpler but adequate solution.

The columns of the matrix are scanned only once, from left to right; local conditions are used to determine if a good place to split the matrix has been encountered. Several options exist for these local conditions. One possibility is to split as soon as a local minimum is reached; i.e., as soon as the number of connections to nodes outside the current block is increased. A second possibility is to use a threshold; this threshold may be supplied by the user. While scanning the matrix from left to right, it is split as soon as the number of connections to nodes outside the current block reaches the threshold. As optimizing to local minima usually turns out not to be best, we have implemented the threshold approach. Further finetuning is possible by taking into account the number of edges in the current block when deciding to apply the local threshold or not; this avoids generating fragments that are 'too small.'

3.3 Linear fragmentation algorithm

In this section we describe an algorithm that fragments a graph in such a way that the fragmentation graph (cf. Sec. 2.1) is guaranteed to be acyclic (i.e., loosely connected). An additional assumption we have to make is that there is some topological information associated to nodes of the graph. For the transportation graphs we are focusing on, this is a reasonable assumption. We will assume that each node has an associated coordinate-pair $\langle x, y \rangle$.

The algorithm starts by selecting a group of start nodes located on an extreme end of the graph. In each iteration, it then accumulates the adjacent edges in a fragment; this idea is illustrated in Fig. 6. Once the number of edges in a fragment has reached a certain threshold (defined as $\mid E \mid / f$, the number of edges divided by the number of fragments we want), the nodes on the boundary are put in a disconnection set and used as starting points for the next fragment. Note that the disconnection sets may become very large using this algorithm. Also note that the fragment size may be unbalanced; in each iteration all edges starting from the boundary nodes have to be added to the frag-

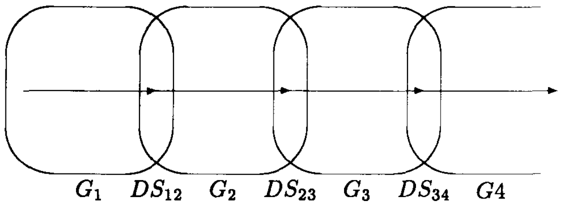

Figure 6: Intuition behind linear fragmentation algorithm

```
/* Input: graph $G = (V, E)$, $f$ = number of fragments */
/* Output: fragments $G_1, \ldots, G_n$; $V = \bigcup_i V_i$; $E = \bigcup_i E_i$
$threshold := | E | / f$;
$start\_n$ := $s$ nodes with smallest $x$-coordinates;
$k := 1$; $E_1 := E_2 := \ldots : E_n := \emptyset$
while $| E | > 0$ do
  while $| E_k | < threshold \wedge | E | > 0$ do
    $new\_e := \{(x,y) \mid x \in start\_n \vee y \in start\_n\}$;
    $start\_n := \{x \mid x \notin V_k \wedge ((x,y) \in new\_e \vee (y,x) \in new\_e)\}$;
    $E_k := E_k \cup new\_e$;
    $E := E \setminus new\_e$
  od;
  $DS_{k(k+1)} := start\_n$;
  $k := k + 1$
od
```

Figure 7: Linear fragmentation algorithm

ment to avoid cycles, we may therefore have fragments that are just the size of the threshold but also fragments that are much larger. The algorithm is shown in Fig. 7

The result of the algorithm is influenced by the choice of the start nodes. Depending on the shape of a graph, a particular choice of start nodes may be preferable. This is illustrated in Fig. 8, where the ellipses represent the same graph, fragmented into 3 fragments. It illustrates that starting on the left side of the graph and going to the right, is preferable to starting at the top and going down. This, because the size of the disconnection sets (nodes on the border between fragments) is much smaller that way. However, graphs often have capricious forms that make a choice of start nodes difficult. We have chosen to start at the leftmost side, but for actual applications we might ask the user to provide us with the start nodes.

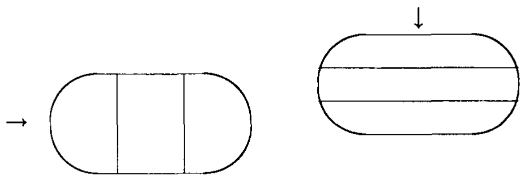

Figure 8: Two ways of starting a fragmentation

4 Experimental results of the algorithms

In the first part of this paper, we have developed a number of algorithms for fragmenting graphs for use by the disconnection set approach. Each of these algorithms focused on a specific goal: equally sized fragments, small disconnection sets, or an acyclic fragmentation graph. In this section we investigate in what respect these goals are achieved. We will show how the algorithms behave w.r.t. different types of graphs, and investigate the characteristics of the fragmentations that result from applying the algorithms we developed.

To test the algorithms, we decided to generate graphs (randomly) with particular characteristics. Then, we tested the algorithms on these graphs to determine the characteristics of the resulting fragmentations. Here, we will report on tests applied to two specific types of graph. First of all, we generated transportation graphs according to the structure previously shown in Fig. 3. Second, we generated graphs without any particular predetermined structure. Our main interest lies with transportation graphs, but testing the algorithms on 'ordinary' graphs may also help to give insight into the performance of our algorithms (in terms of reaching the goals the algorithms were developed for).

We will first discuss how we generated the graphs. Then we will discuss the results of applying our algorithms to these graphs. The results discussed here are derived from [18], which provides more detailed test results.

4.1 Generating graphs

To test the algorithms, we generated graphs in a random way. Input to the generation algorithm were the number of nodes of the graph, the number of fragments that should be generated (in case of transportation graphs), and two parameters for the probability function we used (explained shortly).

As we needed the nodes of the graph to have coordinates for the application of the 'linear fragmentation' algorithm, the first step was to generate coordinates for each node; the coordinates were evenly spread over a given interval. We decided that these coordinates could be of use in generating graphs as well. As our main interest is in transportation graphs—where there exist many connections inside a cluster, but few connecting different clusters—the use of coordinates is particularly appropriate. They can be used to encourage local connections over connections between remote nodes.

In the second step, we generated the edges of the graph. Edges were generated w.r.t. a particular probability function, reflecting the fact that the chances of connections between nodes that are close (in distance) are bigger than the chances of connections between remote nodes. The function we used is the following:

$$P(p,q) = (c_1/n^2)e^{-c_2 d(p,q)}$$

In this function, p and q denote the nodes under consideration, $d(p,q)$ is a function returning the Euclidean distance between p and q, and c_1 and c_2 are the user-provided parameters. By changing c_1 we could influence the number of edges generated (and thereby the connectivity), and by changing c_2 we could influence the probability of generating edges between nodes that are far apart.

For transportation graphs, the abovementioned procedure was first used to generate the required number of fragments. Then, these fragments were connected following the requirements given by the user. Hence, we were able to specify which fragments were connected to each other and by how many edges.

4.2 Fragmenting graphs

After generating graphs we fragmented them using the algorithms described in Sec. 3. We will now present the results of this. The characteristics of the fragmentations that we show are: average size of the fragments F (i.e., number of edges), average size of the disconnection sets DS (i.e., number of nodes), average deviation ΔF from F, and average deviation ΔDS from DS. These characteristics are a good indication in how far the goals set out in Sec. 3 have been met.

4.2.1 Transportation graphs

In Table 1 the results are shown for a number of tests on transportation graphs that have a structure as depicted in Fig. 3. The fragments generated had 25 nodes each and the average number of edges in these graphs was 429; the average number of edges connecting fragments was 2.25.

Algorithm	F	DS	ΔF	ΔDS
center-based	107.25	6.9	21.9	3.6
bond-energy	77.8	2.4	33.8	1.1
linear	78	13.3	24.2	5.3

Table 1: Fragmentation characteristics for transportation graphs, 4 clusters of 25 nodes

From these results we may notice the following. As intended, the bond-energy algorithm fragments the graphs in such a way that the average size of the disconnection sets is smallest (2.4); whereas the linear fragmentation algorithm does not take the size of the disconnection set in account (13.3). We may also note that the variation in size of the fragments is quite large with the bond-energy and linear fragmentation algorithms, whereas the center-based approach achieves a better balance in size of the fragments. Also, the number of fragments is predetermined with the center-based approach, whereas there is a slight variation in number of fragments possible with the other algorithms.

From the experiments we concluded that the center-based approach did not always give the expected results, in particular, it turned out that the disconnection sets were larger than expected and that there was a considerable variation in the size of the fragments. This was due to the way centers were selected; sometimes the selected centers were quite close to each other. As we had already assigned coordinates to each node, we decided to use these coordinates as well in selecting the centers. From now on, we did not select the centers at random from a group of possible centers (remember that these were selected using a weight-function, as explained in Sec. 3.1). Instead, we used the coordinates assigned to the nodes to make sure that the selected nodes would not be too close together.

The results from this change in selecting the centers for the center-based algorithm, is shown in Table 2. These results were obtained for a transportation graph of the same type as discussed before, but this time with 150 nodes per fragment. The number of edges of the graph was 3167. As can be seen from the deviation of the average fragment size, and the average size of the disconnection sets, using the coordinates in selecting the centers gives indeed a considerable im-

Algorithm	F	DS	ΔF	ΔDS
center-based	791.8	69.5	636.3	13.8
distributed centers	791.8	4.3	12.4	2.9

Table 2: Fragmentation characteristics with and without distributed centers, 4 clusters of 150 nodes

Algorithm	F	DS	ΔF	ΔDS
center-based	77	18.1	40.2	8.8
distributed centers	77	18.9	34.7	5.9
bond-energy	93.2	5.4	88.4	2.1
linear	111.8	35.8	42.1	1.25

Table 3: Fragmentation characteristics for general graphs, 1 cluster of 100 nodes

provement.

4.2.2 General graphs

Finally, we tested general graphs, without any superimposed structure as in transportation graphs. Although we developed the algorithms especially with transportation graphs in mind, it is interesting to study their applicability in general.

In Table 3 the results are shown for tests on general graphs existing of 100 nodes and an average number of edges of 279.5. We may note that the algorithms again conform to the idea that underlies them. The bond-energy algorithm fragments a graph in such a way that the disconnection sets are small (5.4), however, the variation in size of the fragments is considerable. The linear fragmentation achieves an acyclic fragmentation, but the size of the disconnection sets is large (35.8). Finally, the center-based algorithms favour an equally balanced workload over small disconnection sets. W.r.t. the center-based algorithms we have to make an additional remark: in the tests we have used the variant that takes the *diameter* of a fragment as an indication for the workload, if we would have taken the other variant that takes *size* of a fragment as indication this would clearly have shown in the results.

4.2.3 Conclusions on experimental results

The results of the experiments show that, both for transportation graphs and for 'ordinary' graphs, the algorithms indeed achieve a fragmentation with the intended characteristics. The variant of the bond-energy algorithm renders a fragmentation with small disconnection sets, at the price of having fragments whose size may differ considerably. The linear fragmentation algorithm renders a loosely connected fragmentation (no cycles in the fragmentation graph), at the price of having large disconnection sets. And the center-based algorithm renders a fragmentation with fragments that are balanced in size, with disconnection sets that are in size somewhere in between the ones resulting from the other algorithms.

From our current perspective, we believe that small disconnection sets will be the main factor in achieving a performance-wise good parallel evaluation of transitive closure queries; therefore, we feel that the variant of the bond-energy algorithm will be most effective when fragmenting a relation. However, the result may very well depend on both the type of graph considered, and on the characteristics of the database system that is being used.

5 Conclusions and future research

We investigated the topic of fragmenting data in such a way that we can achieve a good parallel computation of transitive closure queries. In [10, 11] we introduced a strategy for parallel computation of transitive closure queries, called the disconnection set approach, but we did not yet discuss data fragmentation algorithms. Here we developed and tested the latter.

We indicated the issues that are of main importance when fragmenting data for the disconnection set approach. These issues are: small disconnection sets, equally sized fragments, and an acyclic fragmentation graph. We developed several algorithms, each focusing on a particular characteristic. We focused on a particular class of graphs, so-called transportation graphs, but the algorithms also work in the general case. We presented test results, showing that the results of applying the algorithms indeed conform to the characteristics we required.

If the complexity of the fragmentation graph (describing the way fragments are connected) becomes very high, finding the paths in it that connect the fragment containing the start node with the fragment containing the end node might become computation intensive. In [12] we have described an extension of the disconnection set, called parallel hierarchical evaluation, to cope with that problem. It introduces the concept of a 'high-speed network'; this is a separate fragment that mandatorily has to be traversed when going to a non-adjacent fragment.

Currently, we are undertaking experiments [14] on the PRISMA multi-processor database machine [4, 20]. These experiments will show which of the characteristics identified here, is of main importance when striving for an optimal parallel evaluation of transitive closure queries. This will show which of the algorithms we developed is most useful, and why. It may well be the case that the actual algorithm to be used for data fragmentation depends on the type of graph that is considered, and on the specific characteristics of the underlying database system.

Acknowledgements

We thank Prof. Kees Hoede from the department of Applied Mathematics for sharing with us his ideas on graph-theoretical issues.

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. 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.

[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] AMERICA, P. (Ed.), *Parallel Database Systems, Proc. of the PRISMA Workshop*, LNCS 503, Springer-Verlag, 1991.

[5] CERI, S., CACACE, F., AND HOUTSMA, M.A.W. "An overview of parallel strategies for transitive closure on algebraic machines," in [4].

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

[7] W.T. MCCORMICK, P.J. SCHWEITZER, T. WHITE "Problem decomposition and data reorganization by a clustering technique," in Oper. Res. 20, 5 (Sept.-Oct. 1972), pp. 993–1009.

[8] 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.

[9] C. HOEDE "A new status score for actors in a social network," Technical report MATH-243, University of Twente, 1979.

[10] HOUTSMA M.A.W., APERS P.M.G., AND CERI S. "Distributed transitive closure computation: the disconnection set approach," *Proc. 16th Int. Conf. on Very Large Data Bases*, Brisbane, Aug. 1990, pp. 335–346.

[11] HOUTSMA, M.A.W., APERS, P.M.G., AND CERI, S. "Complex transitive closure queries on a fragmented graph," *Proc. 3rd Int. Conf. on Database Theory*, Lecture Notes in Computer Science, Springer-Verlag, Dec. 1990.

[12] HOUTSMA M.A.W., CACACE F., AND CERI S. "Parallel hierarchical evaluation of transitive closure queries," in *Proc. 1st Int. Conf. on Parallel and Distributed Information Systems*, Miami Beach, Dec. 1991, pp. 130–137.

[13] HOUTSMA, M.A.W. AND APERS, P.M.G. "Algebraic optimization of recursive queries," in *Data and Knowledge Engineering*, 7(4), March 1992.

[14] HOUTSMA, M.A.W., WILSCHUT, A.N., AND FLOKSTRA, J. "Implementation and performance evaluation of a parallel transitive closure algorithm on PRISMA/DB", Technical report INF92-45, University of Twente, June 1992.

[15] HUA, K.A. AND HANNENHALLI, S.S. "Parallel transitive closure computations using topological sort," in *Proc. 1st Int. Conf. on Parallel and Distributed Information Systems*, Miami Beach, Dec. 1991, pp. 122–129.

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

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

[18] SCHIPPER, G.L.V. "Fragmentation design for parallel computation of the transitive closure," M.Sc.-Thesis, University of Twente, Nov. 1991.

[19] VALDURIEZ P. AND S. KHOSHAFIAN "Parallel Evaluation of the Transitive Closure of a Database Relation," in *Int. Journal of Parallel Programming*, 17:1, Feb. 1988.

[20] WILSCHUT, A., FLOKSTRA, J., AND APERS, P.M.G. "Parallelism in a main-memory DBMS: The performance of PRISMA/DB," in *Proc. VLDB*, Aug. 1992.

[21] WOLFSON O. "Sharing the load of logic program evaluation," *Int. Symp. on Database in Parallel and Distributed Systems*, Dec. 1988, pp. 46–55.

[22] WOLFSON O. AND A. OZERI, "A new Paradigm for Parallel and Distributed Rule-processing", in *Proc. ACM-SIGMOD 1990*, pp. 133-142.