Parallel Hierarchical Evaluation of Transitive Closure Queries

Maurice A.W. Houtsma*
Department of Computer Science, University of Twente (email: houtsma@cs.utwente.nl)

Filippo Cacace , Stefano Ceri†
Dipartimento di Elettronica, Politecnico di Milano (email: cacace, ceri @ipmel1.polimi.it)

Abstract

This paper presents a new approach to parallel computation of transitive closure queries using a semantic data fragmentation. Tuples of a large base relation denote edges in a graph, which models a transportation network. We propose a fragmentation algorithm which produces a partitioning of the base relation into several fragments such that any fragment corresponds to a subgraph. One fragment, called high-speed fragment, *collects all edges which guarantee maximum speed, these edges correspond to highways or to high-speed inter-city trains. Thus, the fragmentation algorithm induces a hierarchical relationship between the high-speed fragment and all other fragments. With this fragmentation, any query about paths connecting two nodes can be answered by using just the fragments in which nodes are located and the high-speed fragment. In general, if each fragment is managed by a distinguished processor, then the query can be answered by three processors working in parallel. This schema can be applied recursively to generate an arbitrary number of hierarchical levels.*

1 Introduction

Over the past few years, deductive databases have emerged to bridge the gap between data- and knowledge-base systems. Relational databases are capable of efficiently handling large amounts of data on secondary storage, but their interface is sometimes considered as rather awkward and their expressive power is limited. Therefore, deductive databases offer a logic-based interface, called Datalog, that enables easy formulation of complex queries and, more important, enables formulation of *recursive* queries, on top of a relational system. This has triggered a vast body of research on optimization strategies for recursive queries, both in an algebraic and in a logic context (see, e.g., [4]). A simple, but very important type of recursion is the *transitive closure* operation [11]. Transitive closure operations in algebra amount to the class of linear sirups in Datalog [3], which class has attracted most research on optimization. An example of a transitive closure query, combined with an aggregate computation, is the bill-of-material problem: finding all transitive components of a given part.

An important feature of database technology of the nineties is the use of parallelism for speeding up the execution of complex queries. In particular, *intra-query parallelism* enables the distribution of complex queries to multiple processors. Fragmentation is essential to intra-query parallelism, as it enables a very natural partitioning of query processing. Each processor controls a disk which stores fragments of relations; with this architecture, it is possible to execute selections, projections, and some joins in a distributed way on each processor, and then collect from each processor the result of these operations. Today, a number of research prototypes and a few commercial systems support fragmentation parallelism (e.g., [12, 13]).

The new class of recursive queries can, by its regular structure and complexity of operations, extremely benefit from intra-query parallelism. Therefore, research is now being conducted on parallel execution of recursive queries, both in a logic and in an algebraic context [6, 7, 15]. Some research focuses on the use of hash-based fragmentation [5, 14]; some logic-based approaches focus instead on assigning Datalog rules to processors [10]. An overview of current research on parallel execution strategies for transitive closure is given in [3].

In this paper we generalize the "disconnection set" approach introduced in [7, 8]. In the disconnection

*The research of Maurice Houtsma has been made possible by a fellowship of the Royal Netherlands Academy of Arts and Sciences

†Filippo Cacace and Stefano Ceri are supported by the Esprit project STRETCH and by the CNR project LOGIDATA+

0-8186-2295-4/91 $01.00 © 1991 IEEE

set approach, the semantics of the application domain is used to achieve a significant computation speedup for several types of transitive closure queries: graph reachability, shortest path, and bill-of-material. In this paper we concentrate on the shortest path problem, and we extend and improve on the disconnection set approach. The approach of this paper, called *parallel hierarchical evaluation*, uses a new fragmentation schema that partitions a base relation into several fragments; one of them, called *high-speed fragment*, has special properties. This fragmentation schema allows us to direct queries to specific fragments (in general, 3 fragments are sufficient to solve any shortest path query); this means that relevant data to answer a query are pre-selected. At the same time, each query can be answered in parallel by the processors associated to the relevant fragments; in particular, when each fragment is controlled by one processor, then a query is naturally executed in parallel on three processors.

Our fragmentation algorithm turns out to be familiar to one of the heuristics proposed in [1]. There the issue was to divide a relation into domains in such a way that the search space can be pruned dynamically. However, they do not consider distributed computation; instead they focus on storing precomputed information (in the order of the size of the relation) about the shortest connection between any pair of nodes in a fragment. This leads to a considerable overhead compared to disconnection sets [7]. The assumption made in [1] that domains do not overlap is rather strict and not likely in general; overlapping domains would in their approach lead to a significant effort in trying to bound the search. Although our approach has some characteristics in common with [1], there are significant differences. We give a proper definition of fragments and a full description of a fragmentation algorithm, concentrating on the use of distributed computation for all sorts of transitive closure queries (including e.g. bill-of-material).

The structure of this paper is as follows. In Sec. 2 we give a short description of the disconnection set approach. In Sec. 3 we describe the generalization of this approach to parallel hierarchical evaluation. In Sec. 4 we present an algorithm for fragmentation design[1]. In Sec. 5 we discuss query processing, and in Sec. 6 update management. Finally, in Sec. 7 we draw some conclusions and discuss issues for further research.

2 A primer on disconnection sets

The disconnection set approach, described in [7, 8], uses a fragmentation that is especially tailored to parallel execution of recursive queries. It was suggested by real-world observations concerning travel problems. The basic idea underlying the disconnection set approach is very simple and can be illustrated by considering a railway network in Europe.

Assume that connections in the European railway network are naturally fragmented by nation, and that each fragment is stored on geographically distributed computers that can be accessed through a distributed database system. To find the shortest path from Paris in France to Milano in Italy we may split the question in a number of separate subqueries: find a path from Paris through France to the north-eastern border with Switzerland, then through Switzerland to the Italian border, and finally through Italy to Milano.

This fragmentation leads to a highly selective search process, consisting of determining the properties of connections from the origin to the first border, then between borders of the intermediate fragment, and finally from the last border to the destination city. These queries have the same structure; they apply only to a fragment of the database, and can be executed in parallel.

The relational representation of this is as follows. The connection information is stored into a relation R; each tuple corresponds to an arc of the graph G, which can have cycles. By effect of the fragmentation, 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 $G_i, 1 \leq i \leq n$. *Disconnection sets* DS_{ij} are given by $G_i \cap G_j$.

In order to process queries independently on the fragments, it is required to store some *complementary information* about the identity of border cities (i.e., the nodes in the disconnection set) and the properties of their connections; these properties depend on the particular recursive 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. Such shortest paths may cross the border many times and, therefore, have to be computed and maintained based on the entire graph G; this is discussed in [7]. Complementary information about DS_{ij} is stored together with both fragments G_i and G_j.

In the disconnection set approach, we assume that disconnection sets are much smaller than the fragments, and that they do not overlap (which will in

[1] Although a number of strategies use data fragmentation to achieve parallel execution [5, 14, 15], they typically assume fragmentation based on hash functions, without discussing how to achieve a good fragmentation design.

general be the case). Given these assumptions, the disconnection set approach leads to an effective use of parallel computation on n processors (where n is the number of fragments involved) [7]. This is especially true for fragmentations that are *loosely connected*, i.e., whose graph of components is acyclic; the graph of components has one node for each fragment and one edge for each nonempty disconnection set.

Proofs on the correctness of the disconnection set approach for the various types of transitive closure queries can be found in [8].

3 Parallel Hierarchical Evaluation

In the disconnection set approach, all fragments are "semantically equal." The parallel hierarchical evaluation generalizes the disconnection set approach, by making some of the fragments more equal than others. This distinction is based on real-life observations concerning transport problems.

3.1 Informal Description

If we consider the railway network of many European countries, we note that the countries are subdivided into geographical regions. In each region, a number of slow trains stop at every station; remote regions are connected by inter-city trains that stop only at a few stations, typically the major cities of a region. For long-distance travels, one typically uses the regional network around the departure city in order to reach the high-speed network of inter-city trains, then uses the inter-city trains, and finally uses the regional network around the destination city to arrive at the destination. If arrival and destination are in adjacent regions, an exception to this rule is possible: it might be better to use the slow trains that connect the two regional networks, instead of using the high-speed inter-city trains.

The parallel hierarchical evaluation exactly mimics this intuitive approach to travelling. A *small* subset of the connections is declared to be high-speed; these connections are stored as a separate fragment H. The remaining connections are partitioned into n fragments (corresponding to geographical regions). Each fragment is stored on a separate processor, together with the complementary information for each of its disconnection sets. H is stored on a separate computer, together with the complementary information about the disconnection sets between H and the various other fragments. As in the disconnection set approach, we assume that disconnection sets are small compared to the size of fragments.

Any two fragments are declared as either adjacent or nonadjacent. Two adjacent fragments have a nonempty disconnection set. We build a fragmentation so that the shortest path between any two nodes belonging to fragments G_i and G_j is included within these fragments and within H, but does not include edges from other fragments. In particular, if G_i and G_j are nonadjacent, then the shortest path must include some edge from H.

Note that for achieving a good balance of the work and a profitable parallel computation, it is required that H be small (because it is used in most computations), the fragments be approximately of similar size (even workload), and the disconnection sets be small (to minimize overhead and precomputation).

3.2 Formal Description

We now give a formal description of parallel hierarchical evaluation in terms of the underlying graph and its fragmentation. First we introduce a graph G, its subgraphs G_i, and the high-speed fragment H; DS_{ij} denotes the disconnection set between two generic fragments G_i and G_j, DSH_i denotes the disconnection set between G_i and H. The function W_G is a weight function that assigns a weight to each edge of G.

$$G = (V, E) \qquad W_G : E(G) \rightarrow I\!N$$

$$G_1 = (V_1, E_1) \quad \ldots \quad G_n = (V_n, E_n) \quad H = (V', HS)$$

$$V_1 \cup \ldots \cup V_n \cup V' = V \qquad E_1 \cup \ldots \cup E_n \cup HS = E$$

$$\forall i, j: \quad E_i \cap E_j = \emptyset \qquad \forall i: \quad E_i \cap HS = \emptyset$$

$$DS_{ij} = V_i \cap V_j \qquad DSH_i = V_i \cap V'$$

We assume a function $sh(v_i, v_j)$ that returns the set of arcs constituting the shortest path in G between nodes v_i and v_j. Whenever we mention the shortest path we mean the path for which the summation of the weight of the edges is minimal. We like to design a data fragmentation which satisfies the following property:

Property 3.1 *The shortest path between any two nodes contained in fragments G_h and G_k includes only edges from G_h, G_k, and the high-speed fragment H:*

$$\forall v_i \in V_h, v_j \in V_k, sh(v_i, v_j) \subseteq E_h \cup E_k \cup HS$$

We now introduce the notions of adjacency and non-adjacency.

adjacency Two fragments G_h and G_k, with $h \neq k$, are adjacent if the following properties hold:

1. $\forall v_i \in V_h, v_j \in V_k : sh(v_i, v_j) \subseteq E_h \cup E_k \cup HS$
2. $V_h \cap V_k \neq \emptyset$

non-adjacency Two fragments G_h and G_k, with $h \neq k$, are non-adjacent if the following properties hold:

1. $\forall v_i \in V_h, v_j \in V_k : sh(v_i, v_j) \subseteq E_h \cup E_k \cup HS$
2. $V_h \cap V_k = \emptyset$, with $h \neq k$

The property that enables parallel hierarchical evaluation is the following:

Property 3.2 *Any two pair of fragments G_h, G_k, with $h \neq k$, are either adjacent or non-adjacent.*

Note that Property 3.1 is a logical consequence of Property 3.2, as it is implied by condition 1, common to both adjacency and nonadjacency conditions.

If we postulate that each fragment is managed by a separate processor and is stored together with the complementary information of all its disconnection sets, we may then formulate the following theorem.

Theorem 3.1 *If Property 3.1 holds then the shortest path between any two internal nodes v_i and v_j of fragments G_i and G_j can be computed on three processors in parallel.*

Note that if either of the departure or arrival node fall into the disconnection set between two fragments, then the node(s) must be interpreted as belonging to both fragments; thus, in the worst case (when both departure and arrival nodes are not internal), four of the above computations must be performed; the shortest path is obtained as the minimum of the shortest paths obtained from each computation. Due to our assumptions on the size of disconnection sets, this case is unlikely.

Also note that this formalization allows for trivial solutions, such as declaring the complete graph to be in HS. Obviously, this is not what we want if we aim to achieve parallel computation. When designing the fragmentation we shall generate at least n fragments (with n reasonable large), where the fragments are approximately equal in size.

4 Fragmentation design

In this section we describe how to design a fragmentation that satisfies Property 3.1; fragmentation design is a hard problem, and this section constitutes the core of this paper. Due to space considerations, we only describe the algorithm informally here, a full version is given in [9] The design is initiated by the user's choice of "region centers": the user pre-determines the number of fragments n, and chooses for each fragment the node c_i of the graph which is situated approximately in its center. This initial choice will influence the final outcome and is heuristic in nature. From this choice on, the algorithm develops a fragmentation that satisfies the requirements described at the end of the previous section. Let C denote the set of fragment centers. Fragmentation design is then conducted in five steps.

Step 1. During this step, the shortest path—in terms of the sum of the weight of the edges—between any pair of the nodes in C is computed. All edges belonging to these paths are removed from the graph G and put into the high-speed fragment. Note that the connections in the high-speed fragment need not be acyclic.

Step 2. Then, edges of G are progressively assigned to fragments. This is done by starting from center nodes and by progressively including neighbour nodes and the edges leading to them into the fragments. At each iteration, the next node which is included into a fragment is the unassigned node with minimum distance from a center node—by construction this node can be reached by an edge starting from a node that was already assigned to a fragment. In this way, the fragments being generated have approximately the same diameter (again in terms of the sum of the weight of the edges, not in terms of the number of edges constituting a path). Whenever an edge connects two nodes assigned to different fragments, that edge is marked as "critical" and included into a set D to be examined in step 3. After step 2, each node is assigned to exactly one fragment. The following property holds, where c_i denotes the center of fragment G_i and $sl(v_i, c_i)$ denotes the length of the shortest path between v_i and c_i:

$$\forall i, k : v_k \in V_i, j \neq i, sl(v_k, c_i) \leq sl(v_k, c_j)$$

However, the fundamental Property 3.1 does not hold. As an example of possible property violation, consider Figure 1. Assume that the shortest path between nodes a and d goes through nodes b and c; the shortest path connecting node a of G_1 and d of G_3 thus uses an edge from G_2 and violates Property 3.1. Note that this problem cannot be solved by assigning critical edges (in this case edges (a, b) and (c, d)) to H.

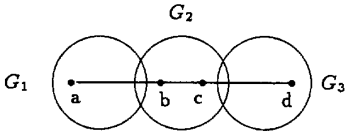

Figure 1: Critical nodes and edges during fragmentation design

Step 3. Step 3 considers "critical edges" in D. We start by building the set B of "boundary nodes" of fragments, defined as the set of nodes from which one critical edge departs:

$$B = \{v_i \mid \exists e_{ij} \in D\}$$

Then, for every pair of nodes v_i and v_j in B, with $v_i \in G_h$ and $v_j \in G_k$, we compute the shortest path that connects them. If this path includes any edge from another fragment G_w ($w \neq h \wedge w \neq k$), then all the edges of this path are inserted into the high-speed fragment H and deleted from D and G_w. After this step, Property 3.1 holds; intuitively, it holds for all pairs of "border nodes" of fragments, by direct construction; but all paths connecting any pair of "interior nodes" have to go through some "border nodes", where they connect to the high-speed fragment, if that is needed.

Although this step is highly combinatorial, its computation is not too hard, since we assume $|B| << |V|$, and we need to consider $|B| \times (|B| - 1)/2$ possible combinations. Section 4.1.2 shows optimizations that apply to Step 3.

Step 4 This step determines whether any two fragments are adjacent or nonadjacent. For all pairs of fragments, the number of critical edges in D that connect them is counted. If this number is below a threshold t, then the fragments are defined as non-adjacent and the edges connecting them are put into the high-speed fragment HS. If the number of connecting edges is greater than t, then fragments are defined to be adjacent, and each connecting edge is arbitrarily assigned to either of them. Eventually, D becomes empty and all edges are assigned to a fragment; this terminates the fragmentation design.

Note that if t is very high, then many pairs of fragments are defined as non-adjacent, and the high-speed fragment is large. Conversely, if t is very low, then many pairs of fragments are defined as adjacent, and the high-speed fragment is small. Query processing is best performed in parallel when most fragments are nonadjacent. The "optimal" situation (i.e., balancing the size of HS, the size of the disconnection sets, and the size of the fragments to achieve a fast response time) for a specific graph may be found by adjusting t.

Step 5. The last step computes the complementary information (see Sec. 2) for the disconnection sets between the adjacent fragments and between each fragment and the high-speed fragment.

Note that this algorithm indeed avoids a trivial solution, and keeps to the guidelines mentioned at the end of Sec. 3.2. In particular, by building fragments where the boundaries are approximately the same distance from the center, and by delaying the assignment of critical edges to fragments, the generated fragments are more or less of equal size.

4.1 Efficient implementation

In this section we discuss the efficient implementation of the hardest steps of the allocation algorithm. Step 4 is computationally easy. Steps 1, 3, and 5 are structurally very similar, as they require solving a large number of shortest path problems. We first address the optimization of Step 2, then we consider Steps 1, 3, and 5 together.

4.1.1 Step 2

In Step 2 of the algorithm we add edges to fragments, based on the minimal distance of nodes to centers. This step is repeated $O(|G|)$ times, and thus must be performed efficiently. Indeed, there is an efficient implementation that greatly reduces its complexity.

We store a list *L1* of ternary tuples (c_i, v_h, d_{ih}), representing that the shortest path from center c_i to node v_h has length d_{ih}. *L1* is sorted based on the third column and represents the partial spanning trees that start from the center nodes in C. We also store a list of edges *L2* that can be joined to either one center c_i or to one node v_h of *L1* and have not yet been included into a spanning tree; *L2* is sorted based on the weight $W_G(e_{hk})$. Initially, *L1* is empty and *L2* includes all tuples of E which survive Step 1 and can be joined with any center c_i, i.e., one of the nodes connected by the edge *is* a center node.

At each iteration, we search for the shortest connection that can be made between an element of *L1* and an element of *L2*, to include edges from *L2* into fragments in increasing order of their distance from the centers. An iteration corresponds to adding one tuple to *L1*, deleting one tuple from *L2*, and possibly moving some tuples from E to *L2*.

An iteration is done in the following way. We take the first element e_{hk} of *L2* and find the first matching tuple (c_i, v_h, d_{ih}) of *L1*; this is a candidate shortest connection, with a distance from the center c_i given by $d_{ik} = d_{ih} + W_G(e_{hk})$; we denote the prefix of the *L1*-list between its top and the matching tuple as *PL1*.

To ensure that the candidate connection is really the one that minimizes the distance of an unassigned edge from a center, we scan the *L2* list forward and search for second match with the *PL1* list. If such a match is found, we compare the candidate shortest connection with the second matching tuple, and possibly exchange them. This process needs to be iterated until either *L2* is completed, or the distance d_{ik} of the current candidate shortest connection is less than the weight of the current element of *L2*.

At the end of this process, the new tuple (c_i, v_k, d_{ik}) is entered into *L1*; e_{hk} is deleted from *L2*; and all edges of E which match with v_k are entered into *L2*. The process continues until E is empty and Step 2 is completed.

4.1.2 Steps 1, 3, and 5

Steps 1, 3, and 5 are structurally very similar; they require computing the shortest path among all pairs of nodes within a set. Step 1 considers as initial set the fragment centers, Step 3 the boundary nodes, and Step 5 the nodes within disconnection sets. Each step then has a complexity $O(i^2)$ times the complexity of solving a shortest path problem in $|G|$, where i equals the number of fragments in Step 1, the number of boundary nodes in Step 3, and the number of nodes in a disconnection set in Step 5; Step 5 has to be repeated for each nonempty disconnection set. The number of boundary nodes is in general much larger than the number of fragments or the size of disconnection sets and therefore dictates the overall complexity of the algorithm. However, to all these steps we can apply the same optimization technique, which reduces the complexity of each step from quadratic to linear.

Let X denote the subset of V for which we want to compute all pairs of shortest paths in G. We select the node x_i of X, and compute all shortest paths departing from x_i by using Dijkstra's algorithm; we recall that the algorithm progressively labels all other nodes of G with the minimum distance from x_i. A version of Dijkstra's algorithm that uses an eXtended Relational Algebra (XRA) and is optimized for relational databases is discussed in [7]. Whenever another node $x_j \in X$ is reached, the shortest path $sh(x_i, x_j)$ becomes available; when all nodes in X have been reached, the construction is arrested, and x_i is deleted. In this way, the complexity of each step becomes linear in the size of X, rather than being quadratic. We recall that in Dijkstra's algorithm constructing all shortest paths from a given node has the same worst case complexity as constructing a single shortest path.

In Step 3, the action to be performed when $sh(x_i, x_j)$ is computed consists in checking that the shortest path is not violating Property 3.1; this does not change Step 3's complexity.

One optimization is specific to Step 3. Each shortest path $sh(x_i, x_j)$ has an upper bound in accumulated weight of the edges given by: $sl(x_i, c_i) + sl(c_i, c_j) + sl(c_j, x_j)$, where c_i denotes the center of the fragment which includes x_i. These numbers are available from Steps 1 and 2. Thus, the computation of shortest paths originating from x_i can be suspended when the greater upper bound of distances between x_i and all nodes in X which have not yet been reached is less than the distance between x_i and the node that was last reached by Dijkstra's algorithm.

5 Query Processing

Suppose that we need to compute the shortest path between nodes v_i in fragment G_h and v_j in fragment G_k. Assume that v_i and v_j are *internal* to G_h and G_k, i.e., they do not belong to any disconnection set.

5.1 Nonadjacent fragments

If fragments G_h and G_k are non-adjacent, we may divide the computation into three independent subqueries:

- From v_i to DSH_h;
- From DSH_h to DSH_k;
- From DSH_k to v_j.

Note that the first subquery is computed by using G_h; the second subquery is computed on the high-speed fragment and the complementary information associated to both DSH_h and DSH_k; the third subquery only requires G_k. (The complementary information can be used in the computation on either fragment or even in the post-processing phase instead; the choice is arbitrary.) The complementary information corresponding to DSH_h and DSH_k can be relationally represented as ternary relations; each of them stores in the first and second column all the combinations of nodes of the disconnection set, and in the third column the distances between them. Each subquery may be computed by running the Dijkstra algorithm on a separate processor; a version of the algorithm which uses extended relational algebra is discussed in [7].

Once the three subqueries are processed, a post-processing phase is needed to choose the shortest path by taking into consideration the various alternative ways of combining the three parts. Let:

- $P1$ denotes a binary relation storing the result to the first subquery; the first column stores the nodes of DSH_h and the second column stores the distance from v_i to them.
- $P2$ denotes a ternary relation storing the result to the second subquery; the first column stores nodes of DSH_h, the second column stores nodes of DSH_k, and the third column stores the distance between them.
- $P3$ denotes a binary relation storing the result to the third subquery; the first column of $P1$ stores the nodes of DSH_k and the second column stores the distance from them to v_j.

Then, in the following relational expression, each tuple corresponds to a different path traversing disconnection sets in all possible ways:

$$T = ((P1 \bowtie_{1=1} P2) \bowtie_{4=1} P3)$$

The following tuple function can be evaluated, adding column 8 to T: $T.8 := T.2 + T.5 + T.7$. The shortest path corresponds to the minimum value of column $T.8$. Note that all these operations are here described relationally, but are really performed in main memory using arrays as data structures.

5.2 Adjacent fragments

If fragments G_h and G_k are adjacent, we also need to consider paths directly connecting them; these are divided into two independent parts:

- From v_i to DS_{hk};
- From DS_{hk} to v_j.

The first part is computed by using G_h and the complementary information associated to DS_{hk}; the second part requires only G_k. (The complementary information can again be used in the computation on either fragment.) Each subquery can be performed independently, hence two processors may be used.

Postprocessing is needed to evaluate the best direct shortest path. In order to do so, we denote $P1$ as the binary relation storing the results of the first subquery and $P2$ as the binary relation storing the results of the second subquery. $P1$ and $P2$ are now processed in a similar way as discussed in the previous subsection. Finally, the best direct shortest path is compared with the best shortest path computed by using the high speed fragment, as discussed in Section 5.2.1.

6 Update Management

Unfortunately, hierarchical fragmentation is very sensitive to updates, and therefore is recommended for base relations which are rather stable. An update is *localized* within a fragment if the following two conditions hold:

1. The tuple which is either added or deleted or updated connects two nodes which are internal to the same fragment G_i.
2. This change does not alter the shortest path between any pair of "border nodes" of the fragment, i.e., nodes included in a disconnection set.

Localized updates can be performed on each fragment, without need of recomputing hierarchical fragmentation. Updates which are *not* localized, however, cannot be dealt with easily[2]. For this reason, we assume that nonlocalized updates should be collected over rather long periods of time, and then applied all together; data distribution should then be re-designed. This update practice is typically used in transportation systems.

7 Conclusions and further research

This paper has presented an approach to the parallel evaluation of shortest path queries on a large base relation. We have discussed the properties that make hierarchical evaluation possible and attractive; we also have discussed how to develop an initial fragmentation that is suited to hierarchical evaluation, how to process queries in parallel, and how to deal with updates. In this paper we have concentrated on the shortest path problem, but our approach can easily be generalized to other problems—such as graph reachability or bill-of-material—by using slightly different complementary information (see [8]).

[2] The apparently easy solution of adding new tuples to the high-speed fragment is unfortunately incorrect; finding a counter-example is an easy exercise, left to the reader.

In the implementation of hierarchical fragmentation, several other optimizations are possible. In particular, the high-speed fragment can be *stored in main memory*, and shortest paths of the high-speed fragments can be *pre-computed and permanently stored.*

The hierarchical approach can be generalized to an arbitrary large number of levels. For instance, the high-speed fragment can be further partitioned into one super-high-speed fragment and several high-speed fragments; this is particularly useful if the high-speed fragment is relatively large. Such generalizations correspond to real-life heterogeneous transport systems (for instance, local transportations, trains, and airplanes). In [1]—where distributed computation was not considered, but similar techniques were used for pruning the search space—such a generalization proved useful for two levels, and assuming that the top levels are small compared to the lower level fragments it would also be useful for multi-level fragmentation. This is exactly what we envision to be the case when we fragment the high-speed network using our fragmentation algorithm.

References

[1] 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.

[2] BANCILHON, F., MAIER, D., SAGIV, Y., AND ULLMAN, J.D. "Magic sets and other strange ways to implement logic programs," in *Proc. ACM SIGMOD-SIGACT Symp. on Principles of Database Systems*, Cambridge, USA, March 1986, pp. 1–15.

[3] CACACE, F., CERI, S., AND HOUTSMA, M.A.W. "An overview of parallel strategies for transitive closure on algebraic machines," in *Proc. Workshop on Parallel Database Systems*, Noordwijk, the Netherlands, Sept. 1990; also appeared as Lecture Notes in Computer Science No. 503, Springer-Verlag, pp. 44–62.

[4] CERI, S., GOTTLOB, G., AND TANCA, L. *Logic programming and databases*, Springer-Verlag 1990.

[5] CHEINEY, J.P. AND DE MANDREVILLE, C. "A parallel strategy for transitive closure using double hash-based clustering," in *Proc. 16th Int. Conf. on Very Large Data Bases*, Brisbane, Australia, Aug. 1990, pp. 347–358.

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

[7] 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, pp. 335–346.

[8] HOUTSMA, M.A.W., APERS, P.M.G., AND CERI, S. "Complex transitive closure queries on a fragmented graph," in *Proc. 3rd Int. Conf. on Database Theory (ICDT'90)*, Lecture Notes in Computer Science, Springer-Verlag, Dec. 1990.

[9] HOUTSMA, M.A.W., CACACE, F., AND CERI, S. "Parallel Hierarchical Evaluation of Transitive Closure Queries," Technical Report 924, University of Twente, the Netherlands, Dec. 1990.

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

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

[12] 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.

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

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

[15] 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.